# Un corso rapido e indolore sul linguaggio Python

Norman Matloff
University of California, Davis
c 2003-2010, N. Matloff

8 Aprile 2010

traduzione italiana di *Andrea Montagner* (12 nov. 2013) sotto licenza FDL

# Contenuti

1 Panoramica ..... 5
1.1 Cosa sono i linguaggi di script? ..... 5
1.2 Perché Python? ..... 5
2 Come utilizzare questo corso ..... 6
2.1 Conoscenze necessarie ..... 6
2.2 Approccio ..... 6
2.3 Quali parti leggere e quando ..... 6
3 Un esempio introduttivo di 5 Minuti ..... 8
3.1 Codice di un programma d'esempio ..... 8
3.2 Liste di Python ..... 8
3.3 Definizione di blocchi Python ..... 9
3.4 Python offre anche una modalità interattiva ..... 10
3.5 Python come calcolatrice ..... 11
4 Un esempio introduttivo di 10 minuti ..... 12
4.1 Codice di un programma d'esempio ..... 12
4.2 Argomenti da linea di comando ..... 13
4.3 Introduzione alla manipolazione dei file ..... 13
5 Dichiarazione (o meno), scopo, funzioni, ecc... ..... 14
5.1 Assenza di dichiarazione ..... 14
5.2 Locali Vs. Globali ..... 14
6 Una coppia di funzioni integrate ..... 15
7 Tipi di variabili/valori ..... 16
7.1 Stringhe Vs Valori numerici ..... 16
7.2 Sequenze ..... 16
7.2.1 Liste (Quasi-Array) ..... 17
7.2.2 Tuple ..... 18
7.2.3 Stringhe ..... 19
7.2.4 Riordino ..... 20
7.3 Dizionari (Hash) ..... 20
7.4 Definizione di funzioni ..... 21
8 Inserimenti da tastiera ..... 23
9 Uso di __name__ ..... 24
10 Programmazione orientata agli oggetti (OOP) ..... 26
10.1 Codice di un programma d'esempio ..... 26
10.2 La parola chiave self ..... 26
10.3 Variabili delle istanze ..... 27
10.4 Variabili delle classi ..... 27
10.5 Costruttori e distruttori ..... 27
10.6 Metodi delle istanze ..... 27
10.7 Metodi delle classi ..... 27
10.8 Classi derivate ..... 28

10.9 Una parola sulla implementazione delle classi ....28
11 L'importanza di comprendere i riferimenti agli oggetti ....30
12 Cancellazione degli oggetti ....32
13 Confronto tra oggetti ....33
14 Moduli e pacchetti ....34
14.1 Moduli ....34
14.1.1 Codice di un programma d'esempio ....34
14.1.2 Come funziona import ....35
14.1.3 Codice compilato ....35
14.1.4 Miscellanea ....35
14.1.5 Una nota sulle variabili globali nei moduli ....36
14.2 Occultamento dei dati ....36
14.3 Pacchetti ....37
15 Gestione delle eccezioni (non solo per eccezioni!) ....39
16 Stringhe di documentazione (docstring) ....40
17 Miscellanea ....41
17.1 Far girare script Python senza invocare esplicitamente l'interprete ....41
17.2 Argomenti denominati nelle funzioni ....41
17.3 Stampa senza a capo o spazi ....41
17.4 Manipolazione delle stringhe formattate ....42
18 Esempio di strutture di dati in Python ....43
18.1 Far uso di idiomi Python ....44
19 Caratteristiche della programmazione funzionale ....46
19.1 Funzioni lambda ....46
19.2 Mappatura ....46
19.3 Filtraggio ....47
19.4 Comprensione delle liste ....48
19.5 Riduzione ....48
A - Scovare gli errori ....49
A.1 Il correttore (debugger) integrato di Python, PDB ....49
A.1.1 Gli elementi di base ....49
A.1.2 Uso delle macro PDB ....51
A.1.3 Usare __dict__ ....52
A.1.4 La funzione type() ....52
A.2 Usare PDB con Emacs ....52
A.3 Caccia agli errori con DDD ....54
A.3.1 Vi servirà uno Speciale PYDB ....54
A.3.2 Avvio e caricamento di DDD ....55
A.3.3 Punti di interruzione ....55
A.3.4 Avvio del vostro codice ....55
A.3.5 Ispezionare le variabili ....55
A.3.6 Miscellanea ....56
A.4 Scovare gli errori con Winpdb ....56

A.5 Scovare gli errori con Eclipse .....56
A.6 Alcuni aiuti interni a Python per la ricerca degli errori .....56
A.6.1 L'attributo dict .....56
A.6.2 La funzione id() .....56
B - Documentazione online .....58
B.1 La funzione dir() .....58
B.2 La funzione help() .....59
B.3 PyDoc .....59
C - Mettere tutte le variabili globali in una classe .....60
D – Sguardo alla macchina virtuale di Python .....61

# 1 Panoramica

## 1.1 Cosa sono i linguaggi di script?

I linguaggi come il C e il C++ permettono ad un programmatore di scrivere il codice ad un livelle molto dettagliato che ha una buona velocità di esecuzione (specialmente nel caso del C). Però nella maggioranza delle applicazioni la velocità di esecuzione non è importante e il molti casi qualcuno potrebbe preferire di scrivere ad un livello superiore. Per esempio, nelle applicazioni che manipolano testi l'unità base in C/C++ è un carattere, mentre per i linguaggi come il Perl e il Python le unità base sono linee di testo e parole nelle linee. Uno può lavorare con linee e parole in C/C++, ma deve compiere uno sforzo maggiore per realizzare la stessa cosa.

Il termine *linguaggio di script* (*scripting language*) non è stato mai definito formalmente, ma qui ci sono le caratteristiche tipiche:

- Uso frequente per l'amministrazione dei sistemi, programmazione Web, elaborazione testi, ecc...
- Molto informale a riguardo della tipicizzazione delle variabili, ad esempio scarsa o nessuna distinzione tra variabili intere, a virgola mobile o stringa. Le matrici [array] possono mischiare elementi di “tipi” differenti, come interi e stringhe. Le funzioni possono restituire non-scalari, ad esempio matrici. I non-scalari possono essere utilizzati come indici per cicli. Ecc...
- Molte operazioni ad alto livello intrinseche al linguaggio, ad es. concatenazione di stringhe e catasta [stack] push/pop.
- Interpretato, piuttosto che venir compilato per l'insieme di istruzioni della macchina ospitante.

## 1.2 Perché Python?

Il primo linguaggio di script veramente popolare è stato il Perl. Tutt'oggi è ancora ampiamente utilizzato, ma i linguaggi del momento sono Python e Ruby, simile al Python. Molta gente, compreso me, preferisce di gran lunga il Python al Perl, siccome è molto più pulito e più elegante. Python è piuttosto popolare presso gli sviluppatori in Google.

I sostenitori di Python, spesso chiamati *pythonisti*, dicono che il Python è così chiaro e così piacevole da scrivere visto che uno potrebbe usarlo per tutti i propri lavori di programmazione, non solo per attività di script. Essi ritengono che sia superiore al C o al C++[1]. Personalmente credo che il C++ sia ridondante e che le sue parti non si colleghino bene insieme; Java è più curato, ma la sua natura fortemente tipizzata è, dal mio punto di vista, fastidiosa e di ostacolo ad una programmazione chiara. Sono rimasto compiaciuto di vedere che Eric Raymond, importante promotore del movimento dell'open source, abbia espresso lo stesso mio punto di vista circa C++, Java e Python.

1 Di nuovo: un'eccezione sarebbero dei programmi realmente bisognosi di una rapida velocità di esecuzione.

# 2 Come utilizzare questo corso

## 2.1 Conoscenze necessarie

Chiunque con anche una minima esperienza di programmazione potrebbe trovare abbastanza accessibile il materiale fino alla Sezione 8.

Il materiale ad iniziare dalla Sezione 10 apparirà abbastanza alla portata di chiunque abbia delle esperienze di linguaggi di programmazione orientata agli oggetti (OOP o Object-Oriented Programming) come il C++ o il Java. Se vi mancano tali conoscenze, potrete ancora essere in grado di leggere queste sezioni, ma probabilmente le scorrerete più lentamente rispetto a coloro che conoscono l'OOP; prestate attenzione soltanto agli esempi, non alla terminologia.

Ci saranno un paio di posti dove descriveremo brevemente delle cose in un contesto Unix, cosicché qualche conoscenza di Unix potrebbe risultare utile, ma certamente non richiesta. Python viene utilizzato anche su piattaforme Windows e Macintosh, non solo Unix.

## 2.2 Approccio

Qui il nostro approccio è diverso da quello della maggioranza dei libri sul Python, o anche della maggior parte dei corsi Web di Python. Il consueto approccio è di trattare dolorosamente tutti i dettagli dall'inizio. Per esempio, l'approccio tradizionale potrebbe essere quello di indicare tutte le forme possibili che può assumere un intero in Python e per questo quante differenti opzioni di linea di comando uno può eseguire con Python.

Io qui evito ciò. Ripeto, l'obbiettivo è di mettere in grado il lettore di acquisire rapidamente delle basi di Python. Costui/costei dovrebbe poi essere capace di approfondire direttamente qualche argomento particolare se e quando ne sorgesse la necessità.

## 2.3 Quali parti leggere e quando

Vorrei suggerirvi di leggere prima fino alla Sezione 8 e poi di mettere voi stessi un po' alla prova con Python. Per prima cosa sperimentate un attimo la modalità interattiva di Python (Sezione 3.4). Successivamente provate a scrivere voi stessi qualche breve programma. Quest'ultimo potrebbe essere un programma completamente nuovo o una semplice modifica di programmi d'esempio presentati qui di seguito[2].

Ciò vi darà una impressione del linguaggio molto più concreta. Se il vostro utilizzo principale del Python sarà per scrivere brevi script e non vorrete usare la libreria del Python, probabilmente questo vi basterà. Tuttavia, molti lettori avranno bisogno di andare più avanti, acquisendo una conoscenza di base delle caratteristiche dell'OOP di Python e dei moduli/pacchetti di Python. Cosicché dovreste successivamente leggere fino alla Sezione 17.

Quello costituirebbe per voi una base molto solida da cui fare buon uso di Python. Alla fine potreste iniziare a notare che molti programmatori di Python fanno uso delle caratteristiche della programmazione funzionale di Python e potreste desiderare di capire ciò che gli altri stanno facendo o di usare voi stessi tali caratteristiche. Se così fosse, la Sezione 19 vi

2 Il file sorgente grezzo di questo corso è scaricabile da http://heather.cs.ucdavis.edu/~matloff/Python/PythonIntro.tex, in modo che non dovete battere da voi stessi i programmi. Potete o modificare una copia di questo file, salvando solo le linee degli esempi di programma che desiderate, oppure utilizzare il vostro mouse per fare operazioni di copia/incolla con le linee che interessano.
Ma se digitate da voi stessi questi esempi, accertatevi di battere esattamente ciò che viene mostrato qui, specialmente le indentazioni. Quest'ultima cosa è cruciale, come diremo più avanti.

permetterà di iniziare.

Non scordate le appendici! Quelle chiave sono le Sezioni A e B.

Io ho anche un certo numero di testi didattici sulla programmazione speciale in Python, ad es. programmazione di rete, iteratori / generatori, ecc... Date un occhiata su **http://heather.cs.ucdavis.edu/˜matloff/python.html**.

# 3 Un esempio introduttivo di 5 Minuti

## 3.1 Codice di un programma d'esempio

Qui c'è un esempio semplice e rapido. Supponete che voglia trovare il valore di

$$g(x)=\frac{x}{1-x^2}$$

per x = 0.0, 0.1, ..., 0.9. Potrei scoprire questi numeri piazzando il codice seguente

```
for i in range(10):
        x = 0.1*i
        print x
        x/(1-x*x)
```

in un file, diciamo **fme.py**, e poi lanciando il programma digitando

```
python fme.py
```

nell'invito [*prompt*] della linea di comando. Il risultato apparirà come questo:

```
0.0
0.0
0.1
0.10101010101
0.2
0.208333333333
0.3
0.32967032967
0.4
0.47619047619
0.5
0.666666666667
0.6
0.9375
0.7
1.37254901961
0.8
2.22222222222
0.9
4.73684210526
```

## 3.2 Liste di Python

Come funziona un programma? Per prima cosa, la funzione **range()** di Python è un esempio di uso delle **liste**, cioè di matrici di Python[3], anche se non abbastanza esplicitamente. Le liste sono assolutamente fondamentali per Python, quindi fate attenzione a quanto segue per le istanze alla parola "list": resistete alla tentazione di trattarla come la parola inglese "list" e pensatela, invece, sempre come il costrutto Python **list**.

La funzione **range()** di Python restituisce un elenco di interi consecutivi, in questo caso l'elenco [0,1,2,3,4,5,6,7,8,9]. Osservate che questa è la notazione ufficiale Python per le liste —una sequenza di oggetti (questi potrebbero essere ogni

3 Io qui parlo di loro liberamente come delle "matrici", ma come potrete vedere, sono più flessibili delle matrici in C/C++. D'altro canto, si può accedere alle vere matrici molto più rapidamente. In C/C++, l'***i*-esimo** elemento di una matrice **X** sono le parole *i* dopo l'inizio della matrice, così possiamo giungere giusto ad esso. Ciò non è possibile con le liste di Python, cosicché le ultime sono più lente da accedere. Il pacchetto aggiuntivo NumPy per Python offre delle vere matrici.

genere di cosa, non necessariamente numeri), separati da virgole e rinchiusi tra parentesi.

Cosicché l'istruzione **for** qui sopra equivale a:

```
for i in [0,1,2,3,4,5,6,7,8,9]:
```

Come potete indovinare ciò determinerà 10 iterazioni [ripetizioni] del ciclo con **i** che è all'inizio 0, poi 1, ecc. Il codice

```
for i in [2,3,6]:
```

ci darebbe tre iterazioni, con **i** che assume i valori 2, 3 e 6.

Python possiede pure un costrutto **while** (sebbene non **until**). C'è anche un'istruzione **break** come quella del C/C++, utilizzata per abbandonare "prematuramente" i cicli. Per esempio:

```
x = 5
while 1:
        x += 1
        if x == 8:
                print x
                break
```

Se non volete fare proprio nulla, c'è pure **pass**, ad. es.:

```
class x(y): pass
```

solo per accertarvi che la classe non sia vuota.

## 3.3 Definizione di blocchi Python

Ora ponete la vostra attenzione su quel apparentemente innocuo due punti alla fine della linea **for**, che definisce l'inizio di un blocco. Diversamente dai linguaggi come C/C++ od anche Perl, che usano le parentesi graffe per definire i blocchi, Python usa una combinazione di due punti e indentazione per definire un blocco. Io sto utilizzando il due punti per dire all'interprete Python,

> Ciao, interprete Python, come stai? Io volevo solo farti sapere, mettendo questo due punti, che inizia un blocco nella prossima linea. Ho indentato quella linea e le due linee che la seguono, più a destra della linea attuale, per dirti che quelle tre linee formano un blocco.

Io scelgo di indentare di tre spazi, ma la quantità non ha importanza fintanto che sono coerente. Se per esempio io avessi scritto[4]

```
for i in range(10):
        print 0.1*i
                print g(0.1*i)
```

l'interprete Python mi avrebbe dato un messaggio di errore, dicendomi che avrei commesso un errore di sintassi[5]. A me è consentito soltanto di indentare un poco a destra con un dato blocco se ho un sottoblocco in quel blocco. Ad es.:

```
for i in range(10):
        if i%2 == 1:
                print 0.1*i
                print g(0.1*i)
```

Qui sto stampando solo i casi in cui la variabile **i** è un numero dispari: **%** è l'operatore "mod" come in C/C++[6]. Nuovamente,

4 Qui **g()** è una funzione che ho definito in precedenza, non mostrata.

5 Tenetelo in mente: i nuovi utenti di Python sono spesso sconcertati da un errore di sintassi derivante da questa situazione.

6 La maggior parte dei consueti operatori C sono presenti in Python, compresi quelli relazionali come **==** visto qui. Qui c'è la notazione **0x** per esadecimale, come c'è ****** del FORTRAN per l'elevamento a potenza. Anche il costrutto **if** può essere accoppiato con **else** come il solito, e potete abbreviare **else if** come **elif**. Gli operatori booleani sono **and**, **or** e **not**. Vedrete degli esempi più avanti. Naturalmente state attenti alle istruzioni Python come **print a or b or c**, in cui la prima espressione vera (ad es. diversa da zero) viene stampata e le altre

notate il due punti alla fine della linea **if** e il fatto che le due linee **print** sono indentate un po' più a destra rispetto alla linea **if**.

Notate pure che, di nuovo a differenza di C/C++/Perl, non ci sono punti e virgola alla fine delle istruzioni del codice del sorgente Python. Una nuova riga significa una nuova istruzione. Se vi serve una linea molto lunga, potete usare il carattere di sbarra invertita per la continuazione, ad es.:

```
x = y + \
        z
```

## 3.4 Python offre anche una modalità interattiva

Una caratteristica veramente carina di Python è la sua capacità di girare in modalità interattiva. Normalmente non lo farete, ma è un gran modo per sperimentare rapidamente qualche funzionalità per vedere come funziona realmente. Ogniqualvolta non siete sicuri se qualcosa funzioni, il vostro motto dovrebbe essere: “Nel dubbio, provalo!” e la modalità interattiva renderà questo rapido e semplice.

Faremo molto di ciò in questa guida, essendo la modalità interattiva un mezzo semplice per illustrare velocemente una funzionalità.

Invece di eseguire questo programma dalla linea di comando in modalità **batch** come abbiamo fatto sopra, potremmo inserire ed avviare il codice in modalità **interattiva**:

```
% python
>>> for i in range(10):
...     x = 0.1*i
...     print x
...     print x/(1-x*x)
...
0.0
0.0
0.1
0.10101010101
0.2
0.208333333333
0.3
0.32967032967
0.4
0.47619047619
0.5
0.666666666667
0.6
0.9375
0.7
1.37254901961
0.8
2.22222222222
0.9
4.73684210526
>>>
```

Qui ho avviato Python che mi ha restituito il suo invito [prompt] interattivo **>>>**. Poi ho cominciato solo a battere il codice, linea per linea. Ogni volta che ero all'interno di un blocco, mi ha dato a tale fine uno speciale invito “...”. Quando ho digitato una linea vuota alla fine del mio codice, l'interprete Python ha compreso che avevo terminato ed ha avviato il codice[7].

In modalità interattiva uno può scorrere su e giù la cronologia dei comandi [*history*] utilizzando i tasti cursori per

ignorate: questo è un comune idioma di Python.

7 La modalità interattiva ci consente di eseguire singole istruzioni Python o di valutare singole espressioni Python. Nel nostro caso qui abbiamo digitato ed eseguito una singola istruzione **for**. La modalità interattiva non è stata concepite affinché noi potessimo battere un programma intero. Tecnicamente potremmo aggirare ciò iniziando con qualcosa come ”if 1:”, rendendo il nostro programma una sola grande istruzione **if**, ma naturalmente non sarebbe in ogni caso conveniente per digitare un programma lungo.

risparmiare battiture.

Per uscire dal Python interattivo, premete ctrl-d.

Stampa automatica: naturalmente, in modalità interattiva, soltanto referenziando o producendo un oggetto, od anche un'espressione, senza assegnarla, causerà la stampa del suo valore, anche senza un'istruzione print. Per esempio:

```
>>> for i in range(4):
...     3*i
...
0
3
6
9
```

Di nuovo: ciò è vero per gli oggetti generici, non solo per le espressioni. Ad es.:

```
>>> open('x')
<open file 'x', mode 'r' at 0xb7eaf3c8>
```

Qui abbiamo aperto il file **x**, che genera un file oggetto. Dal momento che non abbiamo assegnato ad una variabile, diciamo **f**, per un riferimento successivo nel codice, cioè non abbiamo fatto il più consueto

```
f = open('x')
```

l'oggetto è stato stampato. Potremmo ottenere la stessa informazione in questo modo:

```
>>> f = open('x')
>>> f
<open file 'x', mode 'r' at 0xb7f2a3c8>
```

## 3.5 Python come calcolatrice

Fra le altre cose, questo significa che potete usare Python come una rapida calcolatrice (cosa che faccio spessissimo). Se, per esempio, avessi avuto bisogno di sapere quant'è il 5% di 88,88 dollari, avrei potuto digitare

```
% python
>>> 1.05*88.88
93.323999999999998
```

Fra l'altro, è possibile fare delle veloci conversioni tra decimali ed esadecimali:

```
>>> 0x12
18
>>> hex(18)
'0x12'
```

Se mi servono delle funzioni matematiche, devo prima "**import-**are" la libreria matematica di Python. Questo è analogo a ciò che facciamo in C/C++, dove dobbiamo avere una linea **#include** per la libreria nel nostro codice sorgente e dobbiamo collegarlo nel codice macchina per la libreria. Dopo dobbiamo riferirci alle funzioni nel contesto della libreria matematica. Per esempio, le funzioni **sqrt()** e **sin()** devono essere prefissate con **math**[8]:

```
>>> import math
>>> math.sqrt(88)
9.3808315196468595
>>> math.sin(2.5)
0.59847214410395655
```

8 Nella Sez. 14.12 viene mostrato un metodo per evitare il prefisso.

# 4 Un esempio introduttivo di 10 minuti

## 4.1 Codice di un programma d'esempio

Questo programma legge un file di testo specificato nella linea di comando e stampa il numero di righe e parole nel file:

```
1 # legge nel file di testo il cui nome è stato specificato nella linea di comando,
2 # e riporta il numero di righe e parole.
3
4 import sys
5
6 def checkline():
7       global l
8       global wordcount
9       w = l.split()
10      wordcount += len(w)
11
12 wordcount = 0
13      f = open(sys.argv[1])
14 flines = f.readlines()
15 linecount = len(flines)
16 for l in flines:
17      checkline()
18 print linecount, wordcount
```

Diciamo per esempio che il programma sia nel file **tme.py**, e che abbiamo un file di testo **x** con dei contenuti

```
Questo è un
esempio di un
file testuale
```

(Ci sono 5 righe in tutto di cui la prima e l'ultima sono vuote). Se avviamo questo programma con questo file il risultato è:

```
python tme.py_x
5 8
```

Qui, esteriormente, l'aspetto del codice sembra come quello di un programma C/C++: per prima un'istruzione **import**, analoga a **#include** (con il corrispondente collegamento al momento della compilazione) come detto sopra; per seconda la definizione di una funzione; e poi il programma “main” (o principale). Di base questo è un modo corretto di osservarlo, ma tenete in mente che l'interprete Python eseguirà tutto in ordine, iniziando dalla cima. Per esempio, nell'eseguire l'istruzione **import**, che potrebbe realmente comportare l'esecuzione di un qualche codice, se il modulo importato ha un qualche codice in grado di girare per conto proprio. Di più su ciò più avanti. L'esecuzione dell'istruzione **def** per ora non esegue alcun codice, ma l'atto di definire la funzione viene considerato esecuzione.

Ci sono alcune caratteristiche qui in questo programma che non si riscontrano nel primo esempio:

- uso degli argomenti da linea di comando
- meccanismi di manipolazione dei file
- di più sulle liste
- definizione di funzione
- importazione di librerie

- introduzione allo scopo

Tratterò queste funzionalità nelle prossime sezioni.

## 4.2 Argomenti da linea di comando

Per prima cosa spieghiamo **sys.argv**. Python include un **modulo** (cioè libreria) chiamato **sys**, di cui una delle variabili membro è **argv**. Quest'ultima è una lista Python, simile a **argv** in C/C++[9]. L'elemento 0 della lista è il nome dello script, in tal caso **tme.py**, e così via, come in C/C++. Qui nel nostro esempio, in cui avviamo il nostro programma con il file **x**, **sys.argv[1]** sarà la stringa 'x' (le stringhe in Python vengono generalmente specificate con le virgolette). Dal momento che **sys** non viene caricato automaticamente, ci serve la linea **import**.

Sia in C/C++, sia in Python, quegli argomenti da linea di comando sono naturalmente delle stringhe. Se tali stringhe dovessero rappresentare dei numeri, potremmo convertirli. Se avessimo, diciamo, un argomento di un intero, in C/C++ dovremmo eseguire la conversione utilizzando **atoi()**: in Python, dovremmo usare **int()** [10]. Per la virgola mobile in Python dovremmo usare **float()**[11].

## 4.3 Introduzione alla manipolazione dei file

La funzione **open()** è simile a quella in C/C++. La nostra linea

```
f = open(sys.argv[1])
```

ha creato un oggetto della classe **file** e l'ha assegnato a **f**.

La funzione **readlines()** della classe **file** restituisce una lista (tenete in mente che "list" è un termine ufficiale Python) formata dalle righe nel file. Ogni riga è una stringa e tale stringa è un elemento della lista. Siccome qui il file è costituito da 5 righe, il valore restituito chiamando **readlines()** è la lista di cinque elementi

```
['','Questo è un','esempio di','file di testo','']
```

(Sebbene qui non sia visibile, c'è un carattere di fine riga [EOL] in ogni stringa)

9 Non c'è tuttavia bisogno di un analogo **argc**. Python, essendo un linguaggio orientato agli oggetti, tratta le liste come degli oggetti, La lunghezza di una lista viene quindi incorporata in quell'oggetto. Così, se ci serve conoscere il numero di elementi in **argv**, possiamo ottenerlo tramite **len(argv)**.

10 Lo dovremmo anche usare come **floor()** del C/C++ in applicazioni che necessitano di tale operazione.

11 In C/C++ potremmo usare utilizzare **atof()** se fosse disponibile, oppure **sscanf()**.

# 5 Dichiarazione (o meno), scopo, funzioni, ecc...

## 5.1 Assenza di dichiarazione

Le variabili in Python non vengono dichiarate. Una variabile si crea quando viene eseguito il primo assegnamento ad essa. Per esempio, nel programma **tme.py** di cui sopra, la variabile **flines** non esiste finché l'istruzione

```
flines = f.readlines()
```

non viene eseguita.

D'altra parte, una variabile a cui non è stato assegnato un valore ha ancora valore **None** (e questo può essere assegnato ad una variabile, controllato in un'istruzione **if**, ecc...).

## 5.2 Locali Vs. Globali

Python in realtà non possiede delle variabili globale nel senso del C/C++, in cui l'ambito [*scope*] è un programma intero. Tratteremo ciò più avanti nella Sezione 14.1.5, ma per ora supponiamo che il nostro codice sorgente consista di un unico singolo file **.py**: in tal caso, Python avrebbe delle variabili globale piuttosto simili al C/C++.

Python tenta di ricavare l'ambito di una variabile dalla sua posizione nel codice. Se una funzione comprende un qualsiasi codice di assegnazione ad una variabile, allora quella variabile viene ritenuta essere locale . Così, nel codice per **checkline()**, Python supporrebbe che **l** e **wordcount** fossero locali per **checkline()** se noi non lo informassimo in maniera diversa. Otteniamo quest'ultima cosa con l'istruzione **global**.

L'uso di variabili globali qui semplifica l'esposizione e personalmente credo che lo strisciante giudizio critico delle variabili globali sia ingiustificato (v. http://heather.cs.ucdavis.edu/~matloff/globals.html). Infatti in uno dei tipi maggiori di programmazione, i **thread**, l'utilizzo delle globali è praticamente obbligatorio.

Tuttavia, potreste almeno voler raggruppare tutte le vostre globali in una classe, come faccio io. V. Appendice C.

# 6 Una coppia di funzioni integrate

La funzione **len()** restituisce il numero di elementi in una lista, in questo caso, il numero di linee nel file (dal momento che **readlines()** restituisce una lista in cui ciascun elemento consiste in una linea del file).

Il metodo **split()** è un membro della classe **string**[12]. Esso, per esempio, divide una stringa in una lista di parole[13]. Così, ad es., in **checkline()** quando **l** è 'Questo è un' allora la lista **w** sarà uguale a ['Questo','è','un']. (Nel caso della prima riga, che è vuota, **w** sarà uguale alla lista vuota, []).

12 Le funzioni appartenenti a classi sono chiamate *metodi*.
13 L'impostazione predefinita consiste nell'uso di spazi vuoti come criterio di divisione, ma possono si possono usare altri caratteri o stringhe.

# 7 Tipi di variabili/valori

Come consuetudine nei linguaggi di script, il tipo, nel senso di C/C++ **int** o **float** del C/C++, non viene dichiarato in Python. Comunque, l'interprete Python tiene traccia internamente del tipo di tutti gli oggetti. Perciò le variabili di Python non hanno tipi, ma i loro valori sì. In altri termini, una variabile **X** potrebbe essere associata ad un intero in un certo punto del vostro programma e quindi riassociata ad un'istanza di classe in un altro punto. Cioè, Python usa la es **tipizzazione dinamica**.

I tipi di Python comprendono nozioni di scalari, **sequenze** (liste o **tuple**) e dizionari (matrici associative, trattate nella Sez. 7.3), classi, funzioni, ecc.

## 7.1 Stringhe Vs Valori numerici

Diversamente dal Perl, Python distingue tra i numeri e la loro rappresentazione in stringa. Le funzioni **eval()** e **str()** possono essere impiegate per convertire avanti e indietro. Per esempio:

```
>>> 2 + '1.5'
Traceback (most recent call last):
   File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> 2 + eval('1.5')
3.5
>>> str(2 + eval('1.5'))
'3.5'
```

Esistono anche **int()** per convertire da stringhe a interi, e **float()**, per convertire da stringhe a valori a virgola mobile:

```
>>> n = int('32')
>>> n
32
>>> x = float('5.28')
>>> x
5.2800000000000002
```

Consultate anche la Sezione 17.4.

## 7.2 Sequenze

Le liste sono in realtà casi speciali di **sequenze**, che sono tutte tipo matrici ma con alcune differenze. Notate, però, i punti in comune. Tutto di ciò che segue (qualcosa sarà spiegato più avanti) è applicabile a qualsiasi tipo di sequenza:

- l'uso di parentesi quadre per denotare singoli elementi (ad es. **x[i]**)
- la funzione integrata **len()** per fornire il numero di elementi della sequenza[14]
- operazioni di **scomposizione**, cioè di estrazione di sottosequenze
- l'utilizzo degli operatori **+** e ***** per concatenazioni e replicazioni

14 Questa funzione è applicabile pure ai dizionari.

### 7.2.1 Liste (Quasi-Array)

Come affermato prima, le liste sono connotate da parentesi e virgole. Per esempio, l'istruzione

```
x = [4,5,12]
```

imposterebbe **x** con la specificata matrice a 3 elementi.

Le liste possono crescere dinamicamente utilizzando le funzioni **append()** o **extend()** della classe **list**. Per esempio, se, dopo l'istruzione di cui sopra, eseguissimo

```
x.append(-2)
```

ora **x** sarebbe uguale a [4,5,12,-2].

Per le liste sarebbero disponibili altre numerose operazioni, alcune delle quali vengono illustrate nel codice seguente:

```
1     >>> x = [5,12,13,200]
2     >>> x
3     [5, 12, 13, 200]
4     >>> x.append(-2)
5     >>> x
6     [5, 12, 13, 200, -2]
7     >>> del x[2]
8     >>> x
9     [5, 12, 200, -2]
10    >>> z = x[1:3] # "affettamento" della matrice: elementi da 1 a 3-1 = 2
11    >>> z
12    [12, 200]
13    >>> yy = [3,4,5,12,13]
14    >>> yy[3:] # tutti gli elementi che iniziano con indice 3
15    [12, 13]
16    >>> yy[:3] # tutti gli elementi eccetto quelli con indice fino a 3 escluso
17    [3, 4, 5]
18    >>> yy[-1] # significa "entità 1 dal fondo a destra"
19    13
20    >>> x.insert(2,28) # inserisce 28 nella posizione 2
21    >>> x
22    [5, 12, 28, 200, -2]
23    >>> 28 in x # prova di appartenenza: 1 per vero, 0 per falso
24    1
25    >>> 13 in x
26    0
27    >>> x.index(28) # trova l'indice nella lista del valore dato
28    2
29    >>> x.remove(200) # differente da "delete," dal momento che non è indicizzato per valore
30    >>> x
31    [5, 12, 28, -2]
32    >>> w = x + [1,"ghi"] # concatenazione di due o più liste
33    >>> w
34    [5, 12, 28, -2, 1, 'ghi']
35    >>> qz = 3*[1,2,3] # replicazione di lista
36    >>> qz
37    [1, 2, 3, 1, 2, 3, 1, 2, 3]
38    >>> x = [1,2,3]
39    >>> x.extend([4,5])
40    >>> x
41    [1, 2, 3, 4, 5]
42    >>> y = x.pop(0) # cancella e restituisce l'elemento 0
43    >>> y
44    1
45    >>> x
46    [2, 3, 4, 5]
```

Abbiamo pure visto l'operatore **in** in un precedente esempio, utilizzato in un ciclo **for**.

Una lista potrebbe comprendere elementi misti di tipi differenti, comprese altre liste di se stessi.

L'idioma Python include una serie di comuni “trucchi Python” che riguardano le sequenze, ad es. il seguente modo rapido

ed elegante per scambiare due variabili **x** e **y**:

```
>>> x = 5
>>> y = 12
>>> [x,y] = [y,x]
>>> x 12
>>> y 5
```

Liste multidimensionali si possono creare come liste di liste. Per esempio:

```
>>> x = []
>>> x.append([1,2])
>>> x
[[1, 2]]
>>> x.append([3,4])
>>> x
[[1, 2],        [3, 4]]
>>> x[1][1]
4
```

Ma fate attenzione! Osservate cosa può andare storto:

```
>>> x = 4*[0]
>>> y = 4*[x]
>>> y
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
>>> y[0][2]
0
>>> y[0][2] = 1
>>> y
[[0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0]]
```

Il problema consiste nel fatto che l'assegnamento a **y** è stato in realtà una lista di quattro riferimenti alla stessa cosa (**x**). Quando l'oggetto è stato puntato dall'**x** modificato, allora tutte le quattro righe di **y** sono cambiate.

Il Python Wikibook (http://en.wikibooks.org/wiki/Python_Programming/Lists) suggerisce una soluzione nella forma delle **comprensioni delle liste** [*list comprehensions*], che tratteremo nella Sezione 19.4:

```
>>> z = [[0]*4 for i in range(5)]
>>> z
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
>>> z[0][2] = 1
>>> z
[[0, 0, 1, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

### 7.2.2 Tuple

Le **tuple** sono come le liste, ma sono **immutabili**, cioè non modificabili. Sono racchiuse tra parentesi tonde o del tutto prive, piuttosto che da parentesi quadre. Le parentesi sono obbligatorie se sussiste un'ambiguità senza di loro, ad es. negli argomenti delle funzioni. Dovrebbe essere usata una virgola in caso di tupla vuota o singola, ad es. (,) e (5,).

Si possono utilizzare le stesse operazioni, eccettuate quelle che potrebbero cambiare le tuple. Così, per esempio

```
x = (1,2,'abc')
print x[1] # stampa 2
print len(x) # stampa 3
x.pop() # illegale, a causa dell'immutabilità
```

Una funzione **zip()**, che mette insieme i corrispondenti componenti di diverse liste, producendo tuple: ad es.

```
>>> zip([1,2],['a','b'],[168,168])
[(1, 'a', 168), (2, 'b', 168)]
```

### 7.2.3 Stringhe

Le stringhe sono essenzialmente delle tuple di elementi a caratteri. Ma esse sono tra virgolette piuttosto che circondate da parentesi tonde e posseggono una maggiore flessibilità rispetto a quella che avrebbero le tuple di elementi a caratteri. Per esempio:

```
1     >>> x = 'abcde'
2     >>> x[2]
3     'c'
4     >>> x[2] = 'q' # illegale, dal momento che le stringhe sono immutabili
5     Traceback (most recent call last):
6     File "<stdin>", line 1, in ?
7       TypeError: object doesn't support item assignment
8     >>> x = x[0:2] + 'q' + x[3:5]
9     >>> x
10    'abqde'
```

(Potreste meravigliarvi poiché l'ultimo assegnamento

```
>>> x = x[0:2] + 'q' + x[3:5]
```

non viola l'immutabilità. La ragione è che in realtà **x** è un puntatore e noi stiamo semplicemente puntandolo ad una nuova stringa creata da una vecchia. Guardate la Sezione 11).

Come notato, le stringhe sono di più di semplici tuple di caratteri:

```
>>> x.index('d') # come atteso
3
>>> 'd' in x # come atteso
1
>>> x.index('de') # gradita sorpresa
3
```

Come si può vedere, la funzione **index()** dalla classe **str** è stata sovraccaricata, rendendola maggiormente flessibile.

Esistono molte altre pratiche funzioni nella classe **str**. Per esempio, abbiamo visto in precedenza la funzione **split()**. L'opposto di questa funzione è **join()**. Si applica ad una stringa con una sequenza di stringhe per argomento. Il risultato è la concatenazione delle stringhe nella sequenza, con la stringa originale tra ciascuna di esse[15]:

```
>>> '---'.join(['abc','de','xyz']) 'abc---de---xyz'
>>> q = '\n'.join(('abc','de','xyz')) >>> q
'abc\nde\nxyz'
>>> print q
abc
de
xyz
```

Qui c'è qualcosa di più:

```
>>> x = 'abc'
>>> x.upper()
'ABC'
>>> 'abc'.upper()
'ABC'
>>> 'abc'.center(5) # centra la stringa con un insieme di 5 caratteri
' abc '
>>> 'abc de f'.replace(' ','+')
'abc+de+f'
```

Un insieme piuttosto ricco di funzioni per la manipolazione delle stringhe è disponibile anche nel modulo **re** ("regular expression"). La classe **str** è integrata nelle versioni più recenti di Python. Con una versione più datata avrete bisogno di un'istruzione

15 Qui l'esempio mostra il "nuovo" uso di **join()**, ora che i metodi di stringa sono integrati in Python. Guardate più sotto la discussione del "nuovo" contro il "vecchio".

```
import string
```

Quest'ultima classe esiste ancora e la classe più recente **str** non la riproduce del tutto.

### 7.2.4 Riordino

La funzione Python **sort()** si può applicare a qualsiasi sequenza. Per i non scalari, questa fornisce una funzione "confronta", che restituisce un valore negativo, zero o positivo, significante **<**, **=** o **>**. Come spiegazione, ordiniamo una matrice di matrici, usando i secondi elementi quali chiavi:

```
>>> x = [[1,4],[5,2]]
>>> x
[[1, 4],        [5, 2]]
>>> x.sort()
>>> x
[[1, 4],        [5, 2]]
>>> def g(u,v):
...     return u[1]-v[1]
...
>>> x.sort(g)
>>> x
[[5, 2],        [1, 4]]
```

(Ciò potrebbe essere fatto molto più semplicemente usando le funzioni "lambda". Guardate la Sezione 19.1.)

Esiste un modulo di libreria Python, **bisect**, che effettua ricerche binarie con relativi riordini.

## 7.3 Dizionari (Hash)

I dizionari sono **matrici associative**. Il significato tecnico di ciò sarà trattato più avanti, ma da un punto di vista di pura programmazione, ciò vuol dire che uno può impostare delle matrici con degli indici non interi. L'istruzione

```
x = {'abc':12,'navigare':'via'}
```

imposta **x** a quanto ammonta una matrice di due elementi con **x['abc']** che è 12 e **x['navigare']** uguale a 'via'. Noi diciamo che **'abc'** e **'navigare'** sono **chiavi** [*keys*], e 12 e 'via' sono **valori**. Le chiavi possono essere qualsiasi oggetto immutabile, ad es. numeri, tuple o stringhe[16]. L'uso di tuple come chiavi è abbastanza comune nelle applicazioni in Python e dovreste tenere in mente che è disponibile questo apprezzabile strumento.

Internamente, **x** sarebbe qui memorizzato come matrice di quattro elementi e l'esecuzione di un'istruzione come

```
w = x['navigare']
```

richiederebbe all'interprete Python di cercare in quella matrice la chiave 'navigare'. Una ricerca lineare sarebbe lenta e così la memorizzazione interna è organizzata come una tabella di hash. Questo è il motivo per cui l'analogo Perl del concetto di dizionario Python è in realtà chiamato **hash**.

Qui ci sono degli esempi di uso di alcune funzioni appartenenti alla classe **dictionary**:

```
1   >>> x = {'abc':12,'navigare':'via'}
2   >>> x['abc']
3   12
4   >>> y = x.keys()
5   >>> y
6   ['abc', 'navigare']
7   >>> z = x.values()
8   >>> z
```

16 Ora uno scorge il motivo per cui Python distingue tra tuple e liste. Permettere chiavi mutevoli sarebbe un incubo di implementazione e, probabilmente, condurrebbe ad una programmazione soggetta ad errori.

```
9   [12, 'via']
10  x['uv'] = 2
11  >>> x
12  {'abc': 12, 'uv': 2, 'navigare': 'via'}
```

Notate come abbiamo aggiunto un nuovo elemento a **x** vicino alla fine. Le chiavi non occorre che siano delle tuple. Per esempio:

```
>>> x
{'abc': 12, 'uv': 2, 'navigare': 'via'}
>>> f = open('z')
>>> x[f] = 88
>>> x
{<open file 'z', mode 'r' at 0xb7e6f338>: 88, 'abc': 12, 'uv': 2, 'navigare': 'via'}
```

La cancellazione di un elemento da un dizionario si può fare tramite **pop()**: ad es.

```
>>> x.pop('abc')
12
>>> x
{<open file 'x', mode 'r' at 0xb7e6f338>: 88, 'uv': 2, 'navigare': 'via'}
```

L'operatore **in** agisce sulle chiavi del dizionario: ad es.

```
>>> 'uv' in x True
>>> 2 in x
False
```

## 7.4 Definizione di funzioni

Ovviamente l'istruzione **def** si utilizza per definire una funzione. Notate ancora una volta che i duepunti e l'indentazione vengono impiegati per definire un blocco che serve come corpo della funzione. Una funzione può restituire un valore ricorrendo all'istruzione **return**, ad es.

```
return 8888
```

Tuttavia, la funzione non ha un tipo anche se restituisce qualcosa e l'oggetto restituito può essere qualsiasi cosa – un intero, una lista, o altro ancora.

Le funzioni sono **oggetti di prima classe** [*first-class objects*], cioè possono essere assegnati proprio come le variabili. I nomi delle funzioni *sono* delle variabili: possiamo infatti assegnare temporaneamente un insieme di codice ad un nome. Considerate:

```
>>> def quadrato(x):     # definite il codice e puntate la variabile quadrato ad esso
...     return x*x
...
>>> quadrato(3)
9
>>> gy = quadrato        # ora anche gy punta a quel codice
>>> gy(3)
9
>>> def cubo(x):
...     return x**3
...
>>> cubo(3)
27
>>> quadrato = cubo      # puntate la variabile quadrato al codice del cubo
>>> quadrato(3)
27
>>> quadrato = 8.8
>>> quadrato
8.8000000000000007      # non stupitevi del 7
>>> gy(3)               # gy punta ancora al codice del quadrato
```

# 8 Inserimenti da tastiera

La funzione **raw input()** mostrerà un invito [*prompt*] e leggerà ciò che verrà scritto. Per esempio,

```
nome = raw_input('inserisci un nome: ')
```

farebbe apparire “inserisci un nome:”, poi leggerebbe la risposta, indi metterebbe la risposta in **nome**. Notate che l'inserimento dell'utente viene restituito in formato stringa e necessita di essere convertito se consiste in numeri.

Se non volete l'invito, non specificatene uno:

```
>>> y = raw_input()
3
>>> y
’3’
```

In alternativa, potete direttamente specificare **stdin**:

```
>>> import sys
>>> z = sys.stdin.readlines()
abc
de
f
>>> z
[’abc\n’, ’de\n’, ’f\n’]
```

Dopo aver battuto ‘f’, premo ctrl-d per chiudere il file **stdin**.

# 9 Uso di __name__

In alcuni casi è importante sapere se un modulo viene eseguito in proprio o via **import**. Ciò si può determinare attraverso la variabile integrata del Python **__name__**, come di seguito.

Qualsivoglia interprete Python stia girando stia girando, esso viene chiamato **programma di massimo livello** o **top-level program**. Se, per esempio, digitate

```
% python x.py
```

allora il codice in **x.py** è il programma di livello massimo. Se state facendo girare Python in modo interattivo, allora il codice digitato è il programma di livello massimo.

Il programma di livello massimo è conosciuto dall'interprete come **__main__** [*principale*] e il modulo che sta girando in quel momento è indicato come **__name__**. Così, per verificare se un dato modulo stia girando in proprio invece di essere stato importato da altro codice, controlliamo se **__name__** è **__main__**. Se la risposta è sì, vi trovate nel livello massimo e il vostro codice non è stato importato, altrimenti lo è stato.

Per esempio, aggiungiamo un'istruzione

```
print __name__
```

al nostro primissimo esempio di codice, dalla Sezione 3.1, nel file **fme.py**:

```
print __name__
for i in range(10):
        x = 0.1*i
        print x
        print x/(1-x*x)
```

Avviamo due volte il programma. Per prima cosa, lanciamolo da se stesso:

```
% python fme.py
__main__
0.0
0.0
0.1
0.10101010101
0.2
0.208333333333
0.3
0.32967032967
... [resto dell'emissione non mostrato]
```

Ora vediamo cosa accade se lo avviamo dall'interprete interattivo Python:

```
>>> __name__
'__main__'
>>> import fme
fme
0.0
0.0
0.1
0.10101010101
0.2
0.208333333333
0.3
0.32967032967
... [resto dell'emissione non mostrato]
```

La nostra istruzione del modulo

```
print __name__
```

ha stampato **main** la prima volta, ma la seconda ha stampato **fme**.

E' usanza raccogliere il “programma principale” del primo (nel senso del C) in una funzione, tipicamente chiamata **main()**. Così, modifichiamo il nostro esempio soprastante in **fme2.py**:

```
def main():
        for i in range(10):
                x = 0.1*i
                print x
                print x/(1-x*x)

if__name__ == '__main__':
        main()
```

Il vantaggio di ciò è che, quando importiamo questo modulo, il codice non verrà eseguito di seguito. Invece, **fme2.main()** deve essere chiamato, o dal modulo da importare o dall'interprete interattivo. Qui c'è un esempio di quest'ultimo:

```
>>> import fme2
>>> fme2.main()
0.0
0.0
0.1 0.10101010101
0.2
0.208333333333
0.3
0.32967032967
0.4
0.47619047619
...
```

Fra le altre cose, questo sarà un punto essenziale nell'utilizzo degli strumenti di correzione (Sezione A). Pertanto abituatevi ad impostare sempre l'accesso a **main()** in questo modo nei vostri programmi.

# 10 Programmazione orientata agli oggetti (OOP)

Al contrario del Perl, Python è stato orientato ali oggetti sin dall'inizio e così ha un'interfaccia per l'OOP più gradevole, più pulita e più chiara.

## 10.1 Codice di un programma d'esempio

A titolo di spiegazione svilupperemo una classe che lavora con i file di testo. Qui c'è il contenuto del file tfe.py:

```
 1 class textfile:
 2      ntfiles = 0 # conteggio del numero di oggetti file di testo
 3      def __init__(self,fname):
 4              textfile.ntfiles += 1
 5              self.name = fname # nome
 6              self.fh = open(fname) # gestione del file
 7              self.lines = self.fh.readlines()
 8              self.nlines = len(self.lines) # numero di linee
 9              self.nwords = 0 # numero di parole
10              self.wordcount()
11      def wordcount(self):
12              "trova il numero di parole nel file"
13              for l in self.lines:
14                      w = l.split()
15                      self.nwords += len(w)
16      def grep(self,target):
17              "stampa tutte le linee contenenti il target"
18              for l in self.lines:
19                      if l.find(target) >= 0:
20                              print l
21
22 a = textfile('x')
23 b = textfile('y')
24 print "il numero di file di testo aperti è", textfile.ntfiles
25 print "qui ci sono alcune informazioni su di loro (nome, linee, parole):"
26 for f in [a,b]:
27      print f.name,f.nlines,f.nwords
28 a.grep('esempio')
```

In aggiunta al file **x** che ho usato nella soprastante Sezione 4, avevo il file di 2 linee **y**. Qui c'è quello che è successo quando ho avviato il programma:

```
% python tfe.py
il numero di file di testo aperti è 2
qui ci sono alcune informazioni su di loro (nome, linee, parole):
x 5 8
y 2 5
esempio di a
```

## 10.2 La parola chiave self

Diamo un'occhiata alla classe **textfile**. La prima cosa da notare e la prevalenza della parola chiave **self**, che sta a significare l'istanza attuale della classe, analogamente a **this** in C++ e Java[17]. Quindi, **self** è un puntatore alla istanza corrente della classe.

17 In realtà **self** non è una parola chiave. Diversamente da C++/Java, NON DOVETE chiamare questa variabile **self**. Se nella vostra definizione di **init()** stavate per chiamare il primo argomento **me**, e poi per scrivere “me” al posto di “self” in tutta la definizione della classe, ciò avrebbe funzionato bene. Tuttavia, potreste incorrere nell'ira dei puristi pythonisti di tutto il mondo. Perciò non fatelo.

## 10.3 Variabili delle istanze

Nella generica terminologia dell'OOP, una variabile di istanza **x** di una classe è una variabile membro per cui ogni istanza della classe ha un distinto valore di quella variabile. Nel mondo C++ o Java, conoscete ciò come una variabile che non è stata dichiarata **static**. Il termine *variabile di istanza* è il termine generico dell'OOP, non specifico di un linguaggio.

Per vedere come queste funzionino in Python, per prima cosa ricordatevi che una variabile in Python viene creata quando qualcosa le viene assegnato. Quindi, una variabile di istanza in un'istanza di una classe Python non esiste sino a che non le viene assegnata.

Per esempio, quando

```
self.name = fname
```

viene eseguito, la variabile membro **name** viene creata dall'attuale istanza della classe e le viene assegnato il valore indicato.

## 10.4 Variabili delle classi

Una variabile di classe, chiamata **v**, ha un valore comune in tutte le istanze della classe. Di nuovo nel mondo C++ o Java, la conoscete come variabile statica. E' concepita in quanto tale per avere qualche riferimento a **v** nel codice contenuto nella classe ma non in qualsiasi metodo della classe. Un esempio è il codice

```
ntfiles = 0 # conteggio del numero di oggetti di file testuali
```

di cui sopra[18].

Notate che una variabile di classe **v** di una classe **u** è riportata come **u.v** nei metodi della classe e nel codice esterno alla classe. Per il codice interno alla classe, ma non in un metodo, ci si riferisce semplicemente come ad una **v**. Dedicate ora un momento per scorrere il nostro soprastante programma di esempio e osservate esempi di questo con la nostra variabile **ntfiles**.

## 10.5 Costruttori e distruttori

Il costruttore di una classe deve essere chiamato **__init()__**. L'argomento **self** è obbligatorio e ne potete aggiungere altri, come ho fatto in questo caso con un nome di file.

Il distruttore è **__del()__**. Notate che viene invocato solo quando si è creata una collezione di spazzatura, cioè quando tutte le variabili che puntano all'oggetto sono passate.

## 10.6 Metodi delle istanze

Il metodo **wordcount()** è un metodo delle istanze, cioè si applica specificamente al preciso oggetto di questa classe. Di nuovo, nella terminologia C++/Java, questo è un metodo non-**statico**. Diversamente da C++ e Java, in cui **this** è un argomento implicito ai metodi delle istanze, Python rende ampiamente esplicita la relazione: è richiesto l'argomento **self**.

## 10.7 Metodi delle classi

18 Naturalmente, anche se abbiamo collocato quel codice all'inizio della classe, esso potrebbe trovarsi all'inizio della classe, alla fine della classe, o tra due metodi finché non è interno ad un metodo. Nell'ultima situazione **ntfiles** verrebbe considerato una variabile locale nel metodo, non ciò che vogliamo innanzitutto.

Prima della Versione 2.2, Python non aveva mezzi per i metodi delle classi, cioè i metodi che non si applicano a specifici oggetti della classe. Adesso il Python ha due modi (leggermente diversi) per fare ciò, usando le funzioni **staticmethod()** e **classmethod()**. Presenterò l'utilizzo della prima nel seguente miglioramento del codice della Sezione 10.1 nella classe **textfile**:

```
class textfile:
...
def totfiles():
print "il numero totale di file testuali è", textfile.ntfiles totfiles = staticmethod(totfiles)
...
# qui siamo in "main"
... textfile.totfiles()
...
```

Notate che i metodi delle classi non hanno l'argomento **self** (né dovrebbero, giusto?).

Notate pure che questo metodo potrebbe essere chiamato anche se non ci fosse alcuna istanza della classe **textfile**. In questo esempio, dovrebbe venir stampato 0, dal momento che non sono stati contati ancora dei file[19].

## 10.8 Classi derivate

L'ereditarietà costituisce gran parte della filosofia di Python. Un'istruzione come

```
class b(a):
```

inizia la definizione di una sottoclasse **b** di una classe **a**. Si possono realizzare anche ereditarietà multiple, ecc...

Notate che quando viene chiamato il costruttore di una classe derivata, il costruttore della classe base non viene chiamato automaticamente. Se desiderate che venga invocato quest'ultimo, dovete chiamarlo esplicitamente, ad es.

```
class b(a):
        def __init__(self,xinit):       # costruttore della classe b
        self.x = xinit                  # definisce e inizializza una variabile di istanza x
        a.__init__(self)                # chiamata del costruttore della classe base
```

Il tutorial ufficiale di Python nota: “[Nel senso di C++] tutti i metodi in Python sono in effetti virtuali”. Se desiderate estendere, piuttosto che passare oltre un metodo nella classe base, potete riferirvi a quest'ultimo anteponendo il nome della base, come nel nostro esempio sopra **a.__init__ (self)**.

## 10.9 Una parola sulla implementazione delle classi

Una istanza di classe Python viene realizzata internamente come un dizionario. Per esempio, nel nostro programma **tfe.py** soprastante, l'oggetto **b** viene realizzato come un dizionario.

Fra le altre cose, ciò significa che potete aggiungere “al volo” variabili membro ad un'istanza di una classe molto dopo che

---

19 Osservate attentamente che questo è diverso dal valore **None** di Python. Anche se non abbiamo ancora creato istanze della classe **textfile**, il codice

```
ntfiles = 0
```

comunque sarebbe stato eseguito quando avremmo la prima volta avviato l'esecuzione del programma. Come ricordato precedentemente, l'interprete Python esegue il file dalla prima riga in avanti. Quando raggiunge la linea

```
class textfile:
```

esegue poi qualsiasi codice indipendente nella definizione della classe.

è stata creata l'istanza. Stiamo semplicemente aggiungendo un'altra chiave e valore al dizionario. Nel nostro programma “main”, per esempio, potremmo avere un comando come

```
b.name = 'zzz'
```

# 11 L'importanza di comprendere i riferimenti agli oggetti

Una variabile a cui è stato assegnato un valore mutevole in realtà è un puntatore ad un dato oggetto. Per esempio, considerate questo codice:

```
>>> x = [1,2,3] # x è mutevole
>>> y = x # ora x e y puntano entrambi a [1,2,3]
>>> x[2] = 5 # l'oggetto mutevole puntato da x ora "muta"
>>> y[2] # questo significa che anche y[2] cambia in 5!
5
>>> x = [1,2]
>>> y = x
>>> y
[1,2]
>>> x = [3,4]
>>> y
[1,2]
```

Nelle prime poche linee **x** e **y** sono riferimenti ad una lista, un oggetto mutevole. Il comando

```
x[2] = 5
```

cambia poi un solo aspetto di quell'oggetto, ma **x** punta ancora a quell'oggetto. D'altra parte, il codice

```
x = [3,4]
```

ora modifica la stessa **x**, avendo puntato ad un oggetto differente, mentre **y** sta ancora puntando al primo oggetto.

Se nell'esempio precedente avessimo voluto semplicemente copiare la lista referenziata da **x** a **y**, avremmo potuto usare il frazionamento, ad es.

```
y = x[:]
```

Dopo **y** e **x** punterebbero ad oggetti differenti: **x** punterebbe allo stesso oggetto di prima, ma il comando per **y** creerebbe un nuovo oggetto a cui **y** punterebbe. Sebbene quei due oggetti abbiano gli stessi valori nello stesso istante, se l'oggetto puntato da **x** cambia, l'oggetto di **y** non cambierebbe.

Come potete immaginare, ciò diventa delicato quando abbiamo oggetti complessi. Guardate il modulo **copy** di Python per funzioni che svolgeranno copiature di oggetti a vari livelli.

Un simile importante problema sorge con gli argomenti nelle chiamate di funzioni. Qualsiasi argomento che sia una variabile che punti ad un oggetto mutevole può cambiare il valore di quell'oggetto dall'interno della funzione, ad es:

```
>>> def f(a):
...     a = 2*a # i numeri sono immutabili
...
>>> x = 5
>>> f(x)
>>> x
5
>>> def g(a):
...     a[0] = 2*a[0] # le liste sono mutevoli
...
```

```
>>> y = [5]
>>> g(y)
>>> y
[10]
```

I nomi delle funzioni sono anch'essi riferimenti ad oggetti. Ciò che noi consideriamo come il nome di una funzione in realtà è solo un puntatore – uno di quelli mutevoli – al codice di tale funzione. Per esempio,

```
>>> def f():
...     print 1
...
>>> def g():
...     print 2
...
>>> f()
1
>>> g()
2
>>> [f,g] = [g,f]
>>> f()
2
>>> g()
1
```

# 12 Cancellazione degli oggetti

Gli oggetti possono essere cancellati dalla memoria di Python utilizzando **del**: ad es.

```
>>> del x
```

**OSSERVATE CON ATTENZIONE CHE CIO' E' DIVERSO DALLA CANCELLAZIONE DA UNA LISTA O DIZIONARIO**. Se usate **remove()** o **pop()**, per esempio, state semplicemente rimuovendo il puntatore all'oggetto da una data struttura, ma fintanto che c'è almeno un **riferimento**, ad es. un puntatore, ad un oggetto, quell'oggetto occupa ancora spazio in memoria.

Questo può essere il principale problema in programmi di prolungata esecuzione. Se non state attenti a cancellare gli oggetti, o se non sono semplicemente della spazzatura raccolta quando è cessato il loro campo di applicazione, potete accumularne sempre di più ed avere un problema di memoria piuttosto grave. Se vedete la vostra macchina girare sempre più lentamente mentre è attivo un programma, dovreste immediatamente sospettare questo.

# 13 Confronto tra oggetti

Uno può usare l'operatore **<** per confrontare sequenze, ad es.

```
if x < y:
```

per le liste **x** e **y**. Il confronto è lessicografico. Per esempio,

```
>>> [12,'tuv'] < [12,'xyz']
True
>>> [5,'xyz'] > [12,'tuv']
False
```

Naturalmente, dal momento che le stringhe sono sequenze, possiamo confrontare pure loro:

```
>>> 'abc' < 'tuv'
True
>>> 'xyz' < 'tuv'
False
>>> 'xyz' != 'tuv'
True
```

Notate gli effetti di questo, per esempio, sulla funzione **max()**:

```
>>> max([[1, 2], [0], [12, 15], [3, 4, 5], [8, 72]])
[12, 15]
>>> max([8,72])
72
```

Possiamo predisporre confronti su oggetti non sequenziali, ad es. istanze di classi, definendo una funzione **cmp()** nella classe. La definizione inizia con

```
def __cmp__(self,other):
```

Deve essere definita per restituire un valore negativo, zero o positivo in base al fatto che **self** sia minore, uguale o maggiore di **other**.

Un ordinamento piuttosto sofisticato si può realizzare se uno combina la funzione Python **sort()** con una specifica funzione **cmp()**.

# 14 Moduli e pacchetti

Spesso avete sentito che è buona pratica dell'ingegneria del software scrivere il vostro codice in maniera "modulare", cioè spezzarlo in componenti, stile inizio-fine, e rendere il codice "riutilizzabile", cioè scriverlo in modo così generico che voi o qualcun altro possiate farne uso in qualche altro programma. Diversamente da un mucchio di luoghi comuni dell'ingegneria del software seguiti come pecore, questo qui è realmente valido! :-)

## 14.1 Moduli

Un **modulo** è un insieme di classi, funzioni di libreria e così via, tutte in un unico file. A differenza del Perl, non ci sono delle speciali azioni da attuare per rendere modulo un file. Qualsiasi file il cui nome abbia un suffisso **.py** è un modulo[20]!

### 14.1.1 Codice di un programma d'esempio

Per nostra illustrazione, prendiamo la classe **textfile** dal nostro esempio soprastante. Potremmo piazzarlo in un file *separato* **tf.py**, con contenuto

```
1 # file tf.py
2
3 class textfile:
4       ntfiles = 0 # conteggio del numero di oggetti textfile
5       def __init__(self,fname):
6               textfile.ntfiles += 1
7               self.name = fname # name
8               self.fh = open(fname) # gestione file
9               self.lines = self.fh.readlines()
10              self.nlines = len(self.lines) # numero di linee
11              self.nwords = 0 # numero di parole
12              self.wordcount()
13      def wordcount(self):
14              "trova il numero di parole nel file"
15              for l in self.lines:
16                      w = l.split()
17                      self.nwords += len(w)
18      def grep(self,target):
19              "presenta tutte le linee contenenti target"
20              for l in self.lines:
21                      if l.find(target) >= 0:
22                              print l
```

Notate che anche se il nostro modulo qui consiste in un'unica classe , potremmo avere parecchie classi, più variabili globali[21], codice eseguibile non facente parte di una qualche funzione, ecc...).

Il nostro file di programma di prova, **tftest.py**, potrebbe ora assomigliare a questo:

```
1 # file tftest.py
2
3       import tf
4
5       a = tf.textfile('x')
6       b = tf.textfile('y')
7       print "il numero di file testuali aperti è", tf.textfile.ntfiles
8       print "qui c'è qualche informazione su loro (nome, linee, parole):"
9       for f in [a,b]:
```

20 Accertatevi che la parte base del nome del file inizi con una lettera, non diciamo, con una cifra.

21 Sebbene queste sarebbero globali solo per il modulo, non per un programma che importi il modulo. Guardate la Sezione 14.1.5

```
10      print f.name,f.nlines,f.nwords
11      a.grep('esempio')
```

### 14.1.2 Come funziona `import`

L'interprete Python, vedendo l'istruzione **import tf**, potrebbe caricare i contenuti del file **tf.py**[22]. Qualsiasi codice eseguibile in **tf.py** viene quindi eseguito in questo caso

```
ntfiles = 0 # conta il numero di oggetti textfile
```

(Il codice eseguibile del modulo potrebbe essere non solo all'interno di classi. Guardate cosa succede quando facciamo **import fme2** in un esempio del passato Capitolo 9).

Successivamente, quando l'interprete incontra il riferimento a **tf.textfile**, potrebbe cercare un qualcosa nominato **textfile** all'interno del modulo **tf**, cioè nel file **tf.py**, e poi comportarsi di conseguenza.

Un approccio alternativo potrebbe essere:

```
1 from tf import textfile
2
3 a = textfile('x')
4 b = textfile('y')
5 print "Numero di file testuali aperti: ", textfile.ntfiles
6 print "Qui abbiamo alcune informazioni su di loro (nome, linee, parole):"
7 for f in [a,b]:
8       print f.name,f.nlines,f.nwords
9 a.grep('esempio')
```

Ciò fa risparmiare battiture, dal momento che battiamo solo "textfile" al posto di "tf.textfile," rendendo meno disordinato il codice. Tuttavia è probabile che sia meno sicuro (cosa accadrebbe se **tftest.py** avesse qualche altro elemento chiamato **textfile**?) e meno chiaro (l'origine di **textfile** in **tf** potrebbe servire a chiarire le cose in grandi programmi).

L'istruzione

```
from tf import *
```

in questa maniera potrebbe importare tutto in **tf.py**.

In ogni caso, separando la classe **textfile**, abbiamo aiutato a rendere modulare il nostro codice ed, eventualmente, a renderlo disponibile per il riutilizzo.

### 14.1.3 Codice compilato

Come nel caso di Java, l'interprete Python compila qualsiasi codice che esegue in un "**byte code**" (codice a byte) per la macchina virtuale Python. Se il codice è importato, allora il codice compilato viene salvato in un file con suffisso **.pyc**, così da non venire nuovamente compilato più tardi.

Dal momento che i moduli sono oggetti, i nomi delle variabili, funzioni, classi, ecc..., di un modulo sono attributi di quel modulo. Quindi essi sono conservati nel file **.pyc**, e saranno visibili, ad esempio, quando avvierete la funzione **dir()** su quel modulo (Sezione B.1).

### 14.1.4 Miscellanea

Il codice di un modulo (a sé stante, cioè non facente parte di una funzione) viene eseguito immediatamente quando viene

22 Qui nel nostro contesto, probabilmente piazzeremmo i due file nella stessa directory, ma tratteremo più avanti la questione dei percorsi di ricerca.

importato il modulo.

I moduli sono oggetti. Possono essere utilizzati come argomenti di funzioni, restituiscono valori dalle funzioni, ecc... L'elenco **sys.modules** mostra tutti i moduli, anche se importati, del programma attualmente in funzionamento.

### 14.1.5 Una nota sulle variabili globali nei moduli

Python non permette veramente le variabili globali nel senso del C/C++. Un modulo importato di Python non avrà accesso alle variabili globali nel modulo che lo importa, né viceversa.

Per esempio, considerate questi due file, **x.py**,

```
# x.py

import y

def f():
        global x
        x = 6

def main():
        global x
        x = 3
        f()
        y.g()

if __name__ == '__main__': main()
```

e **y.py**:

```
# y.py

def g():
        global x
        x += 1
```

La variabile **x** in **x.py** è visibile in tutto il modulo **x.py**, ma non in **y.py**. Infatti, l'esecuzione della linea

```
x += 1
```

in fondo farà apparire un messaggio d'errore, “global name ’x’ is not defined.” [“il nome globale ’x’ non è definito.”].

In realtà, una variabile globale in un modulo è semplicemente un attributo (cioè un'entità appartenente) di quel modulo, similmente al ruolo di una variabile di una classe all'interno della classe. Quando il modulo B viene importato dal modulo A, il nome di spazio di B viene copiato in quello di A. Se il modulo B ha una variabile globale **X**, allora il modulo A creerà una variabile con quel nome, il cui valore iniziale sarà qualunque il modulo B aveva per la sua variabile con quel nome al momento dell'importazione. Ma modifiche a **X** in uno dei moduli NON avrà riscontro nell'altro.

Diciamo che **X** si modifichi in B, ma vogliamo che il codice in A sia in grado di ottenere l'ultimo valore di **X** in B. Possiamo fare ciò includendo in B una funzione, chiamata, poniamo, **GetX()**. Supponiamo che A abbia importato tutto da B, quindi A avrà una funzione **GetX()** che è una copia della funzione di B con quel nome, e il cui unico scopo è di restituire il valore di **X**. A meno che B modifichi quella funzione (cosa che *è* possibile, ad esempio possono essere assegnate delle funzioni), le funzioni nei due moduli saranno sempre le stesse e quindi A potrà usare la sua funzione per avere il valore di **X** in B.

## 14.2 Occultamento dei dati

Python non ha una solida forma di occultamento dei dati paragonabile a quella di **private** ed altri simili costrutti in C++. Offre una ridotta dotazione di questo genere, anche se:

Se anteponete un trattino basso (underscore) al nome di una variabile in un modulo, non sarà importato se verrà utilizzata la forma **from** di **import**. Per esempio, se nel modulo **tf.py** in Sezione 14.1.1 fosse contenuta una variabile **z**, allora un'istruzione

```
from tf import *
```

significherebbe che **z** è accessibile solo come **z**, piuttosto che **tf.z**. Se, d'altro canto avessimo nominato questa variabile **z**, allora l'istruzione precedente non avrebbe reso accessibile questa variabile come **z**: avremmo dovuto usare **tf.z**. Naturalmente la variabile sarebbe ancora visibile dall'esterno del modulo, ma con la richiesta del prefisso **tf.** avremmo evitato la confusione con variabili chiamate in maniera simile nel modulo di importazione.

Un doppio trattino basso determina una storpiatura, con un altro trattino basso più il nome del file anteposto.

## 14.3 Pacchetti

Come menzionato precedentemente, uno potrebbe piazzare più di un'unica classe in un dato modulo, se le classi sono strettamente correlate. Una generalizzazione di ciò si manifesta quando uno ha diversi moduli che sono collegati. I loro contenuti potrebbero non essere così strettamente connessi da metterli semplicemente tutti assieme in un solo modulo gigantesco, ma potrebbero avere ancora una relazione abbastanza stretta da volerli raggruppare in qualche altro modo. Ciò è dove salta fuori la nozione di **pacchetto** [o **package**].

Per esempio, potete scrivere alcune librerie che hanno a che fare con qualche programma Internet che avete scritto. Potreste avere un modulo **web.py,** con classi che avete scritto per programmi che effettuano accessi Web, e un altro modulo **em.py** che serve per il programma di e-mail. Invece di combinarli assieme in un solo grande modulo, potreste tenerli come file separati nella stessa directory, diciamo **net**.

Per rendere questa directory un pacchetto, piazzate semplicemente un file **init.py** in essa. Il file può essere vuoto, oppure può essere usato in modo più sofisticato per alcune operazioni d'avvio.

Per importare questi moduli, dovreste usare istruzioni come

```
import net.web
```

Ciò dice all'interprete Python di cercare un file **web.py** all'interno della directory **net**. Quest'ultima, o più precisamente, la genitrice di quest'ultima, deve trovarsi nel vostro percorso di ricerca del Python. Se, per esempio, l'intero nome di percorso di **net** fosse

```
/u/v/net
```

allora la directory **/u/v** avrebbe necessità di essere nel vostro percorso di ricerca del vostro Python. Sevi trovate in un sistema Unix e state usando la shell C, per esempio, potreste digitare

```
setenv PYTHONPATH /u/v
```

Se avete diverse directory speciali come questa, mettetele tutte assieme usando i due punti come delimitatori:

```
setenv PYTHONPATH /u/v:/aa/bb/cc
```

Il percorso corrente è contenuto in **sys.path**. Nuovamente, consiste in un elenco di stringhe, una stringa per ciascuna directory, separate dai due punti. Può essere stampato o modificato dal vostro codice proprio come qualsiasi altra

variabile[23].

Le directory dei pacchetti hanno spesso delle sottodirectory, sottosottodirectory e così via. Ognuna può contenere un file **init.py**.

23 Ricordatevi: dovete prima importare **sys**.

# 15 Gestione delle eccezioni (non solo per eccezioni!)

Naturalmente, le funzioni integrate e di libreria del Python non hanno un codice di ritorno degli errori in stile C per verificare se hanno avuto successo. Si utilizza, piuttosto, il meccanismo di gestione delle eccezioni di Python **try/except**, ad es.

```
try:
        f = open(sys.argv[1])
except:
        print 'open failed:',sys.argv[1]
```

Qui c'è un altro esempio:

```
try:
        i = 5
        y = x[i]
except:
        print 'no such index:', i
```

Ma l'idioma Python usa questo per del codice che non sta funzionando in un contesto di eccezioni. Diciamo, per esempio, che vogliamo trovare l'indice del numero 8 nella lista **z** e, se non c'è tale numero, agiungerlo per prima cosa alla lista. Potremmo farlo in questo modo:

```
try:
        place = x.index(8)
except:
        x.append(8)
        place = len(x)
```

Come visto sopra, utilizzate **try** per verificare un'eccezione e **raise** per sollevarne una.

# 16 Stringhe di documentazione (docstring)

Esiste una stringa tra i doppi apici, “trova il numero di parole nel file”, all'inizio di **wordcount()**. Questa viene chiamata *docstring* o *stringa di documentazione*. Serve come un tipo di commento, ma durante l'esecuzione, in modo da poter essere utilizzata da debugger e simili. Consente, pure, agli utenti che hanno solo la forma compilata del metodo, diciamo un prodotto commerciale di accedere ad un "commento". Qui c'è un esempio di come accedere ad essa, utilizzando il precedente **tf.py**:

```
>>> import tf
>>> tf.textfile.wordcount.__doc__
’trova il numero di parole nel file’
```

Normalmente una stringa di documentazione occupa diverse linee. Per creare questo genere di stringa, usate triple virgolette ["""]. Il metodo **grep()** è un altro metodo di istanza, questo con un argomento oltre a **self**.

Naturalmente, gli argomenti dei metodi in Python si possono passare solo per valore, nel senso del C. Le funzioni possono avere degli effetti collaterali rispetto al parametro se quest'ultimo è un puntatore (Python non possiede dei formali puntatori, ma ha riferimenti: v. Sezione 11).

Notate pure che quel **grep()** fa uso di una delle numerose operazioni su stringhe di Python, **find()**. Ricerca la stringa dell'argomento all'interno della stringa oggetto, restituendo l'indice della prima ricorrenza della stringa dell'argomento dentro la stringa oggetto, oppure -1 se non ne trova nessuna[24].

24 Le stringhe si possono trattare anche come liste di caratteri. Per esempio, 'geometria' può essere considerata come una lista di 9 elementi e applicando **find()** alla sottostringa ’met’ ritornerebbe 3.

# 17 Miscellanea

## 17.1 Far girare script Python senza invocare esplicitamente l'interprete

Poniamo che abbiate uno script Python **x.py**. Finora abbiamo trattato di come farlo girare tramite il comando[25]

```
% python x.py
```

Ma se indicate la posizione dell'interprete Python nella prima riga di **x.py**, ad es.

```
#! /usr/bin/python
```

e usate il comando Unix **chmod** per rendere eseguibile **x.py**, allora potete avviare **x.py** digitando semplicemente

```
% x.py
```

Ciò è necessario, per esempio, se state invocando il programma da una pagina web.

Meglio ancora, potete far sì che Unix ricerchi nel vostro ambiente la posizione di Python mettendo questo nella prima linea di **x.py**:

```
#! /usr/bin/env python
```

Ciò è più portabile dal momento che piattaforme differenti possono collocare il Python in directory differenti.

## 17.2 Argomenti denominati nelle funzioni

Considerate questo piccolo esempio:

```
1 def f(u,v=2):
2       return u+v
3
4 def main():
5       x = 2;
6       y = 3;
7       print f(x,y) # stampa 5
8       print f(x) # stampa 4
9
10 if __nome__ == '__main__': main()
```

Qui l'argomento **v** viene chiamato **argomento denominato** [**named argument**], con **valore predefinito** 2. L'argomento "normale" **u** è detto a **mandatory argomento obbligatorio** [**mandatory argument**], dal momento che deve essere specificato al contrario di **v**. Altro termine per **u** è **argomento posizionale** [**positional argument**], perché il suo valore viene ricavato dalla sua posizione nell'ordine di dichiarazione degli argomenti della funzione. Gli argomenti obbligatori devono essere dichiarati prima degli argomenti denominati.

## 17.3 Stampa senza a capo o spazi

Una istruzione **print** stampa automaticamente un carattere di a capo [newline]. Per eliminarlo, aggiungete una virgola. Per esempio:

25 Questa sezione sarà specifica di Unix

```
print 5, # stampato ancora nulla
print 12 # ora stampato '5 12', con fine riga [end-of-line]
```

L'istruzione **print** separa automaticamente gli elementi con degli spazi. Per sopprimere gli spazi, utilizzate l'operatore di concatenazione delle stringhe, **+**, e possibilmente la funzione **str()**, ad es.

```
x = 'a'
y = 3
print x+str(y) # stampa 'a3'
```

Naturalmente, **str(None)** è **None**.

## 17.4 Manipolazione delle stringhe formattate

Python supporta "printf()" in stile C, ad es.

```
print "i fattori di 15 sono %d e %d" % (3,5)
```

stampa

```
i fattori di 15 sono 3 e 5
```

Notate l'importanza di scrivere '(3,5)' piuttosto di '3,5'. Nell'ultimo caso l'operatore **%** penserebbe che il suo operando sia solo 3, mentre gli serve una tupla da 2 elementi. Ricordiamo che le parentesi che racchiudono una tupla possono essere omesse purché non vi sia alcuna ambiguità, ma qui non è quello il caso.

Questo è carino, ma è di molto più potente, rispetto alla sola stampa, per la manipolazione in generale delle stringhe. In

```
print "i fattori di 15 sono %d e %d" % (3,5)
```

la porzione

```
"i fattori di 15 sono %d e %d" % (3,5)
```

è un'operazione su stringa, che produce una nuova stringa: il **print** stampa semplicemente quella nuova stringa. Per esempio:

```
>>> x = "vecchio di %d anni" % 12
```

La variabile **x** è ora la stringa 'vecchio di 12 anni'.

Questo è un idioma molto comune, abbastanza potente[26].

26 Alcuni programmatori C/C++ potrebbero riconoscere la similitudine con **sprintf()** della C library.

# 18 Esempio di strutture di dati in Python

Sotto c'è una classe Python per realizzare un albero binario. I commenti dovrebbero rendere il programma autoesplicativo (senza giochi di parole)[27].

```
1 # bintree.py, un modulo per gestire alberi binari ordinati: i valori da memorizzare
2 # possono essere generici, fintanto che esiste una relazione di ordinamento
3
4 # qui abbiamo solo routine da inserire e stampare, ma potremmo aggiungere cancellazione,
5 # ecc...
6
7 class treenode:
8       def __init__(self,v):
9               self.value = v;
10              self.left = None;
11              self.right = None;
12      def ins(self,nd): # inserisce il nodo nd nelle radici dell'albero in self
13              m = nd.value
14              if m < self.value:
15                      if self.left == None:
16                              self.left = nd
17                      else:
18                              self.left.ins(nd)
19              else:
20                      if self.right == None:
21                              self.right = nd
22                      else:
23                              self.right.ins(nd)
24      def prnt(self): # stampa le radici del sottoalbero in self
25              if self.value == None: return
26              if self.left != None: self.left.prnt()
27              print self.value
28              if self.right != None: self.right.prnt()
29
30 class tree:
31      def __init__(self):
32              self.root = None
33      def insrt(self,m):
34              newnode = treenode(m)
35              if self.root == None:
36                      self.root = newnode
37                      return
38              self.root.ins(newnode)
```

E qui c'è una prova:

```
1 # trybt1.py: prova di bintree.py
2 # uso: python trybt.py numeri_da_inserire
3
4 import sys
5 import bintree
6
7       def main():
8       tr = bintree.tree()
9       for n in sys.argv[1:]:
10              tr.insrt(int(n))
11      tr.root.prnt()
12
13 if __name__ == '__main__': main()
```

Una buona cosa del Python è che possiamo utilizzare nuovamente lo stesso codice per oggetti non numerici, purché siano confrontabili (Ripassate la sezione 13). Così possiamo fare la stessa cosa con le stringhe, usando le classi **tree** e **treenode**

27 Ma avete compreso il gioco di parole?

COSÌ COME SONO, SENZA MODIFICHE, ad es.

```
# trybt2.py: prova di bintree.py

# uso: python trybt.py stringhe_da_inserire

import sys
import bintree

def main():
        tr = bintree.tree()
        for s in sys.argv[1:]:
                tr.insrt(s)
        tr.root.prnt()

if __name__ == '__main__': main()

% python trybt2.py abc tuv 12
12
abc
tuv
```

O anche

```
# trybt3.py: prova di bintree.py

import bintree

def main():
        tr = bintree.tree()
        tr.insrt([12,'xyz'])
        tr.insrt([15,'xyz'])
        tr.insrt([12,'tuv'])
        tr.insrt([2,'y'])
        tr.insrt([20,'aaa'])
        tr.root.prnt()

if __name__ == '__main__': main()

% python trybt3.py
[2, 'y']
[12, 'tuv']
[12, 'xyz']
[15, 'xyz']
[20, 'aaa']
```

## 18.1 Far uso di idiomi Python

Nell'esempio della Sezione 10.1 è meglio porre una particolare attenzione alla linea

```
for f in [a,b]:
```

dove **a** e **b** sono oggetti di tipo **textfile**. Ciò spiega il fatto che gli elementi all'interno di una lista non devono essere scalari[28]. Ancor più importante, spiega che l'utilizzo davvero efficace Python significa stare lontani dai classici cicli ed espressioni con elementi di matrici in stile C. Questo è ciò che rende il codice più pulito, chiaro e ed elegante. E' dove Python brilla particolarmente.

Non dovreste usare quasi mai cicli **for** in stile C/C++ - ad es. dove un indice (poniamo **j**) viene controllato nei confronti di un limite superiore (diciamo **j < 10**) e incrementato al termine di ciascuna iterazione (poniamo **j++**).

Naturalmente, In effetti, potrete evitare spesso dei cicli espliciti e e dovreste fare così ogni volta che è possibile. Per esempio, il codice

```
self.lines = self.fh.readlines()
```

28 Né tutti devono essere proprio dello stesso tipo. Uno può avere diversi elementi all'interno della stessa lista.

```
self.nlines = len(self.lines)
```

in quello stesso programma è molto più pulito di quello che noi avremmo, diciamo, in C. In quest'ultimo, avremmo bisogno di impostare un ciclo che leggesse nel file una sola riga per volta, incrementando una variabile **nlines** ad ogni iterazione del ciclo[29].

Un altro grande modo per evitare cicli è di utilizzare le caratteristiche della **programmazione funzionale** del Python, descritta nella Sezione 19.

Il ricorso all'uso di idiomi Python è spesso indicato dai *pythonisti* come il modo "*pythonico*" di fare le cose.

29 A proposito, osservate il riferimento ad un oggetto all'interno di un oggetto, **self.fh**.

# 19 Caratteristiche della programmazione funzionale

Queste caratteri forniscono modi concisi di fare cose che, sebbene sicuramente fattibili tramite costrutti più elementari, compattano il vostro codice e quindi lo rendono piè facile da scrivere e leggere. Esse possono pure far girare molto più velocemente il codice. Inoltre, ciò può aiutarci ad evitare errori, dal momento che una grossa parte dell'infrastruttura che avremmo bisogno di scrivere noi stessi (che sarebbe soggetta ad errori), si prenderà automaticamente cura di noi attraverso i costrutti della programmazione funzionale.

Qui, ad eccezione della prima caratteristica (funzioni lambda), queste caratteristiche eliminano la necessità di esplicitare cicli e riferimenti a elementi di liste. Come menzionato nella Sezione 18.1, ciò produce un codice più pulito, più chiaro.

## 19.1 Funzioni lambda

Le **funzioni lambda** forniscono un modo per definire delle brevi funzioni. Vi aiutano ad evitare di ingombrare il vostro codice con un sacco di "singole righe" che vengono chiamate solo una volta. Per esempio:

```
>>> g = lambda u:u*u
>>> g(4)
16
```

Noate con attenzione che questo NON è un utilizzo tipico delle funzioni lambda: era solo per illustrare la sintassi. Di solito le funzioni lambda non sarebbero definite in modo a se stante come sopra: piuttosto, verrebbero definite all'interno di altre funzioni come **map()** e **filter()**, come vedremo prossimamente.

Qui c'è un'illustrazione più realistica, che si rifa all'esempio di ordinamento della Sezione 7.2.4:

```
>>> x = [[1,4],[5,2]]
>>> x
[[1, 4],        [5, 2]]
>>> x.sort()
>>> x
[[1, 4],        [5, 2]]
>>> x.sort(lambda u,v: u[1]-v[1])
>>> x
[[5, 2],        [1, 4]]
```

E' necessaria una piccola spiegazione. Se guardate l'aiuto in linea di **sort()**, troverete che la definizione sarà

```
sort(...)
        L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*;
        cmp(x, y) -> -1, 0, 1
```

Vedete che il primo elemento è un argomento denominato (richiamate la Sezione 17.2), **cmp**. Quella è la nostra funzione di confront, che abbiamo definito sopra come `lambda u,v: u[1]-v[1]`.

La forma generale di una funzione lambda è

```
lambda arg 1, arg 2, ...: espressione
```

Così, sono ammessi argomenti multipli, ma il corpo stesso della funzione deve essere un'espressione.

## 19.2 Mappatura

La funzione **map()** converte una sequenza in un altra, applicando la medesima funzione a ciascun elemento della sequenza. Per esempio:

```
>>> z = map(len,["abc","clouds","rain"])
>>> z
[3, 6, 4]
```

Così abbiamo evitato di scrivere un esplicito ciclo **for**, con il risultato di un codice che è un po' più pulito, semplice da scrivere e leggere[30].

Nell'esempio sopra abbiamo usato una funzione integrata, **len()**. Avremmo potuto usare anche nostre funzioni: spesso queste vengono opportunamente indicate come funzioni lambda, ad es.:

```
>>> x = [1,2,3]
>>> y = map(lambda z: z*z, x)
>>> y
[1, 4, 9]
```

La condizione che il corpo di una funzione lamba consista solo in un'espressione è piuttosto limitante, non consentendo, per esempio, costrutti if-then-else. Se voleste veramente avere quest'ultimi, potreste usare un artificio. Ad esempio, per implementare qualcosa come

```
if u > 2: u = 5
```

potremo agire come segue:

```
>>> x = [1,2,3]
>>> g = lambda u: (u > 2) * 5 + (u <= 2) * u
>>> map(g,x)
[1, 2, 5]
```

Chiaramente, ciò non è fattibile se non in situazioni semplici. In casi maggiormente complessi, potremmo usare una funzione non-lambda. Per esempio, qui c'è una versione rivisitata del programma in Sezione 4.1:

```
 1 import sys
 2
 3      def checkline(l):
 4      global wordcount
 5      w = l.split()
 6      wordcount += len(w)
 7
 8 wordcount = 0
 9      f = open(sys.argv[1])
10 flines = f.readlines()
11 linecount = len(flines)
12 map(checkline,flines) # sostituisce il vecchio ciclo 'for'
13 print linecount, wordcount
```

Notate che **l** ora è un argomento di **checkline()**.

Naturalmente, ciò potrebbe essere ridotto ancor di più, modificando il cuore del programma "main" soprastante in

```
map(checkline,open(sys.argv[1]).readlines)
```

Ma questo diventerebbe più difficile da leggere e correggere.

## 19.3 Filtraggio

La funzione **filter()** funziona come **map()**, a parte che elimina gli elementi della sequenza che soddisfano una determinata condizione. La funzione a cui viene applicato **filter()** deve essere valutata come booleana, cioè deve restituire il valore vero

30 Nota a parte: osservate nuovamente che, se se non l'avessimo assegnata a **z**, la lista [3,6,4] sarebbe stata stampata comunque. In modalità interattiva, qualsiasi istruzione Python di non-assegnmento stampa il valore del risultato.

o falso desiderato. Per esempio:

```
>>> x = [5,12,-2,13]
>>> y = filter(lambda z: z > 0, x)
>>> y
[5, 12, 13]
```

Questo, di nuovo, ci permette di evitare di scrivere un ciclo **for** e un'istruzione **if**.

## 19.4 Comprensione delle liste

Questo vi permette di compattare un ciclo **for** che produce una lista. Per esempio:

```
>>> x = [(1,-1), (12,5), (8,16)]
>>> y = [(v,u) for (u,v) in x]
>>> y
[(-1, 1), (5, 12), (16, 8)]
```

Questo è più compatto del primo poiché inizializza **y** a [], avendo poi un ciclo **for** in cui chiamiamo **y.append()**.

Diventa ancora migliore quando fatto in forma annidata. Diciamo, ad esempio, di avere una lista di liste che vogliamo concatenare assieme, ignorando il primo elemento di ciascuna. Qui c'è cime dovremmo farlo utilizzando le comprensioni delle liste:

```
>>> y
[[0, 2, 22], [1, 5, 12], [2, 3, 33]]
>>> [a for b in y for a in b[1:]]
[2, 22, 5, 12, 3, 33]
```

## 19.5 Riduzione

La funzione **reduce()** viene usata per applicare la somma o altre operazioni simil-aritmetiche ad una lista. Per esempio,

```
>>> x = reduce(lambda x,y: x+y, range(5))
>>> x
10
```

Qui naturalmente **range(5)** è [0,1,2,3,4]. Ciò che fa **reduce()** è, per prima cosa, aggiungere i primi due elementi di [0,1,2,3,4], cioè con 0 che riveste il ruolo di **x** e 1, che riveste il ruolo di **y**. Ciò dà una somma di 1. Quindi quella somma ricopre il ruolo di **x** e il successivo elemento di [0,1,2,3,4], 2, ricopre il ruolo di **y**, producendo una somma di 3, ecc... Infine **reduce()** termina il suo lavoro e restituisce il valore 10.

Ancora una volta, questo ci ha consentito di evitare un ciclo **for**, più un'istruzione in cui avremmo inizializzato **x** a 0 prima del ciclo **for**.

# A - Scovare gli errori

NON andate a caccia di errori semplicemente aggiungendo e togliendo delle istruzioni **print**. Utilizzate uno strumento di debug! Se non siete dei regolari utenti di strumenti di debug, allora state causando a voi stessi una sofferenza inutile e tempo sprecato: guardate la mia presentazione sulla ricerca degli errori su

`http://heather.cs.ucdavis.edu/~matloff/debug.html`

## A.1 Il correttore (debugger) integrato di Python, PDB

Il correttore integrato di Python, PDB, è piuttosto primitivo, ma è molto importante per comprendere come funziona per due ragioni:

- PDB viene utilizzato indirettamente da strumenti di correzione più sofisticati. Una buona conoscenza di PDB accrescerà la vostra abilità a usare quegli altri strumenti.
- Qui vi mostrerò come aumentare l'utilità di PDB anche come correttore a se stante.

Sotto presenterò PDB in una sequenza di forme via via più utili:

- La forma base.
- La forma base incrementata da un uso strategico di macro.
- La forma base in congiunzione con l'editor di testo Emacs.
- La forma base in congiunzione con la DDD GUI per correttori.

### A.1.1 Gli elementi di base

Dovreste essere in grado di trovare PDB nella sottodirectory **lib** del vostro pacchetto Python. In un sistema Unix, per esempio, ciò probabilmente si trova in qualcosa come **/usr/lib/python2.2**. **/usr/local/lib/python2.4**, ecc... Per correggere uno script **x.py**, battete

```
% /usr/lib/python2.2/pdb.py x.py
```

(Se **x.py** avesse avuto argomenti da linea di comando, essi sarebbero stati posti dopo **x.py** nella linea di comando).

Naturalmente, dal momento che userete parecchio PDB, sarebbe meglio creargli un alias. Per esempio, sotto Unix nella shell C:

```
alias pdb /usr/lib/python2.2/pdb.py
```

così che potrete scrivere più semplicemente

```
% pdb x.py
```

Una volta che vi troviate in PDB, impostate il vostro primo punto di interruzione [*breakpoint*], diciamo alla linea 12:

```
b 12
```

Potete renderlo anche condizionale, per esempio

```
b 12, z > 5
```

Battete **c** (“continua”), che si otterrà in **x.py** e poi si fermerà al punto di interruzione. Quindi continuate come il solito con le principali operazioni che sono come quelle di GDB:

- **b** (“break”) per impostare un punto di interruzione
- **tbreak** per impostare un punto di interruzione per una sola volta
- **ignore** per specificare che un certo punto di interruzione verrà ignorato le prossime k volte, dove k viene specificato nel comando
- **l** (“list”) per elencare alcune linee del codice sorgente
- **n** (“next”) per saltare alla prossima linea, non fermandosi nel codice della funzione se l'attuale linea è una chiamata a funzione
- **s** (“subroutine”) simile a n, ad eccezione che si accede alla funzione in caso di chiamata
- **c** (“continua”) per continuare fino al successivo punto di interruzione
- **w** (“where”) per ottenere un rapporto sullo stack
- **u** (“up”) per muoversi di un livello in su nello stack, ad esempio per cercare là una variabile locale
- **d** (“down”) per muoversi di un livello in giù nello stack
- **r** (“return”) continua l'esecuzione fino al ritorno della funzione attiva
- **j** (“jump”) per saltare ad altra linea *senza* che sia eseguito il codice nel mezzo
- **h** (“help”) per ottenere un (minimo) aiuto in linea (ad esempio **h b** per avere un aiuto sul comando **b** e semplicemente **h** per ottenere un elenco di tutti i comandi: battete **h pdb** per avere un tutorial su PDB[31]
- **q** (“quit”) per uscire da PDB

Con l'avvio di PDB, otterrete il suo invito, o *prompt* (`Pdb`).

Se avete un programma multi-file, i punti di interruzione possono essere specificati nella forma breakpoints can be specified in the form `module name:line number`. Per esempio, supponete che il vostro modulo principale sia **x.py**, e che importi **y.py**. Impostate un punto di interruzione alla linea 8 dell'ultimo come segue:

```
(Pdb) b y:8
```

Notate, tuttavia, che non potete fare ciò fino a che **y** è stato realmente importato da **x**[32].

Quando state eseguendo PDB, state eseguendo Python nella sua modalità interattiva. Perciò, potete dare qualsiasi comando Python nell'invito PDB. Potete definire variabili, chiamare funzioni, ecc... Ciò può essere altamente utile.

Per esempio, sebbene PDB includa il comando **p** per la stampa dei valori di variabili ed espressioni, di solito non è necessario. Per vedere perché, ripensate che ogni volta avviate Python in modalità interattiva, semplicemente battendo il nome

31 Il tutorial funziona attraverso un impaginatore [*pager*]. Battete la barra spaziatrice per andare alla pagina successiva e il tasto q [*quit*] per terminare.

32 Notate pure che, se il modulo fosse implementato in C, non sareste naturalmente in grado di interromperlo lì.

di una variabile o espressione comporterà la stampa di essa—esattamente ciò che **p** avrebbe fatto, senza digitare la 'p'.

Così, se **x.py** contiene una variabile **ww** e avviate PDB, invece di battere

```
(Pdb) p ww
```

potete semplicemente digitare

```
ww
```

e il valore di **ww** verrà stampato sullo schermo[33].

Se il vostro programma ha una complicata struttura di dati, potreste scrivere una funzione per stampare sullo schermo tutta o parte di quella struttura. Poi, dal momento che PDB vi permette di fornire qualsiasi comando nell'invito PDB, potreste semplicemente chiamare questa funzione in quel invito, così da avere una stampa più sofisticata, specifica dell'applicazione.

Dopo che il vostro programma finisca sotto PDB o cada in un errore di esecuzione, potete riavviarlo senza uscire da PDB - importante, dal momento che non volete perdere i vostri punti d'interruzione – battendo semplicemente **c**. E sì, se avete modificato il vostro codice sorgente da allora, il cambiamento si rifletterà in PDB[34].

Se date a PDB un comando a passo singolo come **n** quando siete in una linea Python che svolge operazioni multiple, dovrete fornire il comando **n** più volte (o impostare un punto di interruzione temporaneo per scavalcare ciò).

Per esempio,

```
for i in range(10):
```

fa due operazioni. Per prima cosa chiama **range()** e poi imposta **i**, cosicché dovreste fornire **n** due volte. E che ne dite di questo?

```
y = [(y,x) for (x,y) in x]
```

Se **x** ha, poniamo, 10 elementi, allora dovrete dare il comando **n** 10 volte! Qui vorrete certamente impostare un punto di interruzione per evitarlo.

### A.1.2 Uso delle macro PDB

L'innegabile natura scarna di PDB può essere rimediata un pochino facendo buon uso del comando **alias**, che io suggerisco fortemente. Per esempio, battete

```
alias c c;;l
```

Ciò significa che ogni volta che continuate, alla successiva fermata in un punto di interruzione, otterrete automaticamente un listato del codice vicino. Questo farà realmente molto per compensare l'assenza di una GUI di PDB.

Infatti, ciò è così importante che dovreste inserirlo nel vostro file di inizializzazione di PDB, che in Unix si trova in **$HOME/.pdbrc**[35]. In questo modo l'alias sarà sempre disponibile. Potreste fare lo stesso per i comandi **n** e **s**:

```
alias c c;;l
```

33 Tuttavia, se il nome della variabile è lo stesso di quello di un comando di PDB (o sua abbreviazione), l'ultimo avrà la precedenza. Se, per esempio, avete una variabile **n**, battendo poi **n**, risulterà che verrà eseguito il comando **n[ext]** di PDB, piuttosto che esserci una stampa del valore della variabile **n**. Per ottenere quest'ultima avreste dovuto battere **p n**.

34 PDB è, come visto sopra, soltanto un programma Python esso stesso. Quando riavviate, esso reimporterà il vostro codice sorgente. Ovviamente, la ragione per cui vengono mantenuti i vostri punti di interruzione è che naturalmente essi sino variabili in PDB. In particolare, essi vengono conservati in variabile utente denominata **breaks** nella classe **Pdb** in **pdb.py**. Quella variabile viene definita come un dizionario, con le chiavi che sono nomi dei vostri file sorgente **.py**, e con gli elementi che sono le liste dei punti di interruzione.

35 Python cercherà quel file anche nella vostra attuale di directory.

```
alias n n;;l
alias s s;;l
```

C'è pure un comando **unalias**, per cancellare un alias.

Potete scrivere altre macro che siano specifiche di un particolare programma che state correggendo. Per esempio, supponiamo nuovamente che abbiate una variabile denominata **ww** in **x.py**, e che desiderate controllare il suo valore ogni volta che il correttore va in pausa, diciamo nei punti di interruzione. Quindi cambiate l'alias soprastante in

```
alias c c;;l;;ww
```

### A.1.3 Usare __dict__

Nella Sezione A.6.1 sottostante, mostreremo che, se **o** è un oggetto di una qualche classe, allora stampare **o.__dict__** stamperà tutte le variabili membri di quell'oggetto. Di nuovo, potrete combinare queste con le capacità degli alias di PDB, ad es.

```
alias c c;;l;;o.__dict__
```

In realtà, sarebbe più semplice e più generico utilizzare

```
alias c c;;l;;self
```

In questo modo ottenete informazioni sulle variabili membro, non importa che in che classe siete. D'altro canto, apparentemente ciò non produce informazioni sulle variabili membro nella classe genitrice.

### A.1.4 La funzione type()

Nel leggere codice di qualcun altro, o anche uno proprio, potrebbe non essere chiaro a che tipo di oggetto realmente si riferisca una variabile. Per questo, la funzione **type()** è talvolta utile. Qui ci sono alcuni esempi del suo uso:

```
>>> x = [5,12,13]
>>> type(x)
<type 'list'>
>>> type(3)
<type 'int'>
>>> def f(y): return y*y
...
>>> f(5)
25
>>> type(f)
<type 'function'>
```

## A.2 Usare PDB con Emacs

Emacs è una combinazione di editor di testo e collezione di strumenti. Molti ingegneri del software metterebbero la mano sul fuoco per esso. E' disponibile per Windows, Mac e Unix/Linux: è ricompreso nella maggior parte delle distribuzioni Linux. Ma anche se non siete degli *aficionados* di Emacs, potete scoprire che è un modo eccellente per usare PDB. Potete suddividere Emacs in due finestre, una per editare il vostro programma e l'altra per PDB. Come passate al vostro codice nella seconda finestra, potete vedere da voi stessi i progressi attraverso il codice nella prima .

Per partire, poniamo nel vostro **x.py**, andate alla finestra di comando (qualunque abbiate sotto il vostro sistema operativo), e digitate o

```
emacs x.py
```

oppure

```
emacs -nw x.py
```

Il primo creerà una nuova finestra Emacs, in cui avrete a disposizione le operazione tramite mouse, mentre l'ultimo avvierà Emacs nella finestra corrente con operazioni solo testuali. Chiamerò il primo “modalità GUI”.

Battete poi **M-x pdb**, dove nella maggior parte dei sistemi “M”, che sta per “meta”, significa il tasto Escape (o Alt) piuttosto che la lettera M. Vi verrà chiesto come eseguire PDB: rispondete nella maniera in cui eseguirete PDB esternamente a Emacs (ma con un intero nome del percorso), ad es.

```
/usr/local/lib/python2.4/pdb.py x.py 3 8
```

dove il 3 e l'8 in questo esempio sono argomenti a linea di comando del vostro programma.

A quel punto Emacs si suddividerà in due finestre, come descritto in precedenza. Potete impostare punti di interruzione nella finestra PDB come al solito, oppure battendo **C-x space** alla linea desiderata nella finestra del vostro programma: qui e sotto , “C-” significa battere il tasto Control e tenerlo premuto mentre premete il tasto successivo.

A quel punto, avviate PDB come il solito.

Se modificate il vostro programma e state utilizzando la versione GUI di Emacs, battete **IM-Python | Rescan** per far conoscere a PDB la nuova versione del vostro programma.

Oltre a coordinare PDB con il vostro errore, notate che un altro vantaggio di Emacs in questo contesto è che Emacs si troverà in modalità Python, che vi darà alcuni comandi extra di modifica specifici di Python. Li descriverò in seguito.

In termini du comandi generali di modifica, inserite “Emacs tutorial” o “Emacs commands” nel vostro motore di ricerca preferito e vedrete tonnellate di risorse. Qui ve ne fornirò solo abbastanza per iniziare.

Qui per prima c'è la nozione di **buffer**. Ogni file che state modificando[36] ha il proprio buffer. Pure qualsiasi altra azione che intraprendete produce un buffer. Per esempio, se invocate uno dei comandi di aiuto on-line di Emacs, un buffer viene generato per esso (che potete modificare, salvare, ecc..., se lo desiderate). Qui un esempio rilevante è PDB. Quando scrivete **M-x pdb**, ciò produce un buffer per esso. Così, in un certo momento, potete avere diversi buffer. Potreste ache avere molteplici finestre, sebbene qui per semplicità supporremo di averne solo due.

Nella tabella seguente mostriamo comandi per entrambe le versioni di Emacs, quella esclusivamente testuale e quella GUI. Naturalmente potete usare i comandi testuali anche nella GUI.

36 Ce ne potrebbero essere diversi alla volta, ad es., se il vostro programma consiste in due o più file sorgente.

| azione | testo | GUI |
|---|---|---|
| movimento cursori | tasti direzione, Pagina Su/Giù | mouse, scrollbar sinistra |
| annullare | C-x u | Edit \| Undo |
| tagliare | C-spazio (mossa cursore) C-w | selezionare regione \| Edit \| Cut |
| incollare | C-y | Edit \| Paste |
| ricerca di stringa | C-s | Edit \| Search |
| selezionare regione | C-@ | selezionare regione |
| andare ad altra finestra | C-x o | cliccare finestra |
| ingrandire finestra | (1 linea per volta) C-x ˆ | trascinare barra |
| ripetere i comandi seguenti n volte | M-x n | M-x n |
| listare buffer | C-x C-b | Buffers |
| andare in un buffer | C-x b | Buffers |
| uscire da Emacs | C-x C-c | File \| Exit Emacs |

Durante l'utilizzo di PDB, tenete in mente che il nome del vostro buffer PDB inizierà con “gud”, ad es. **gud-x.py.**

Potete ottenere un elenco delle operazioni speciali di Python in Emacs digitando **C-h d** e poi richiedendo informazioni in **modalità python** (***python-mode***). Una cosa carina appena fuori dal normale è che la **modalità python** di Emacs aggiunge un tocco speciale all'autoindentazione: farà automaticamente indentare ulteriormente subito dopo una linea di **def** or **class**. Qui ci sono alcune operazioni:

| azione | testo | GUI |
|---|---|---|
| regione di commento | C-spazio (mossa cursori) C-c # | selezionare regione \| Python \| Commento |
| andare ad inizio def o class | ESC C-a | ESC C-a |
| andare a fine def o class | ESC C-e | ESC C-e |
| andare in blocco esterno | C-c C-u | C-c C-u |
| spostare regione a destra | marcare regione, C-c C-r | marcare regione, Python \| Shift right |
| spostare regione a sinistra | marcare regione, C-c C-l | marcare regione, Python \| Shift left |

## A.3 Caccia agli errori con DDD

DDD, disponibile su molti sistemi Unix (e liberamente scaricabile se il vostro sistema non ce lo avesse), è una GUI per parecchi debugger, come GDB (per C/C++), JDB (per Java), Perl debugger integrato nel Perl, e così via. Può essere utilizzato nel PDB per Python, rendendovi più piacevole e produttivo l'utilizzo di PDB.

### A.3.1 Vi servirà uno Speciale PYDB

Il primo impiego di DDD su PDB è stato progettato da Richard Wolff. Egli ha modificato leggermente **pdb.py** per questo uso, chiamando il nuovo PDB **pydb.py**. Quest'ultimo è stato ideato per Python 1.5, ma me ne ha ha gentilmente fornito un aggiornamento. Vi serviranno i file

```
http://heather.cs.ucdavis.edu/~matloff/Python/DDD/pydb.py
http://heather.cs.ucdavis.edu/~matloff/Python/DDD/pydbcmd.py
http://heather.cs.ucdavis.edu/~matloff/Python/DDD/pydbsupt.py
```

Piazzate i file da qualche parte nel vostro percorso di ricerca, poniamo **/usr/bin**. Assicuratevi di avergli attribuito il permesso di esecuzione e che **bdb.py** della libreria di Python si trovi nel vostro **PYTHONPATH**.

### A.3.2 Avvio e caricamento di DDD

Per iniziare, diciamo a scovare gli errori in **fme2.py** della Sezione 9, per prima cosa assicuratevi che **main()** sia impostato come descritto in tale sezione.

Quando invocate DDD, ditegli di usare PYDB:

```
ddd --debugger /usr/bin/pydb.py
```

Poi nella Console di DDD, cioè la sottofinestra dei comandi di PDB (vicino al fondo), digitate

```
(pdb) file fme2.py
```

In seguito, quando apportate una modifica al vostro codice sorgente, date ancora il comando

```
(pdb) file fme2.py
```

I vostri punti di interruzione dell'ultimo avvio verranno conservati.

Selezionate Program | Run come il solito per impostare i vostri argomenti della linea di comando, e poi avviate (potreste ottenere la comparsa di una finestra d'errore “DDD: No Source”, ma battete solo OK e ignoratela).

### A.3.3 Punti di interruzione

Per definire un punto di interruzione [*breakpoint*], cliccate il tasto di destra da qualche parte nello spazio vuoto della linea nel vostro file sorgente e scegliete Set Breakpoint (o Set Temporary Breakpoint, oppure Continue to Until Here, se del caso).

### A.3.4 Avvio del vostro codice

Per avviare, cliccate su Program | Run [Avvio Programma], inserite i vostri argomenti nella linea di comando del programma se non c'è niente nella casella Run with Arguments [Avvio con Argomenti], e cliccate Run [Avvio] in quella finestra a scomparsa. Vi verrà chiesto di battere Continue [Continuare], che potreste fare cliccando Program | Run | [Avvio | Programma], ma conviene farlo cliccando Continue [Continuare] nella piccola finestra sommario dei comandi (ma non usate qui Run [Avvio]).

Potete poi cliccare su Next [Prossimo], Step [Passo], Cont, ecc. Il marcatore della linea di esecuzione corrente è disegnato come una 'I', anche se piuttosto sbiadita quando il puntatore del mouse non si trova nella sezione di codice sorgente della finestra DDD.

Naturalmente, non fate riferimento a **sys.argv** nel codice presente all'interno di una classe. Quando il vostro programma viene caricato per la prima volta, qualsiasi codice presente verrà eseguito, e poiché non sono stati ancora caricati gli argomenti della linea di comando, otterrete un errore “*index out of range*”. Evitatelo inserendo del codice che richiami **sys.argv** sia all'interno di una funzione nella classe, sia completamente fuori della classe.

### A.3.5 Ispezionare le variabili

Potete ispezionare il valore di una variabile spostando il puntatore del mouse su qualsiasi istanza della variabile nella finestra del codice sorgente.

Come ricordato nella Sezione A.1.3, se **o** è un oggetto di una qualche classe, allora stampare **o.__dict__** produrrà tutte le variabili membro di questo oggetto. In DDD potete fare ciò ancora più facilmente nel modo seguente. Mettete semplicemente questa espressione in un commento, ad es.

```
# o.__dict__
```

e quindi ogni qualvolta vogliate ispezionare le variabili membro di **o**, muovete semplicemente il puntatore del mouse su tale espressione nel commento!

Fate buon uso della caratteristica di DDD che permette di mostrare in modo continuo una variabile. Cliccate semplicemente con il tasto di sinistra su qualsiasi istanza della variabile e poi scegliete Display [Mostra].

### A.3.6 Miscellanea

DDD, sviluppato originariamente per il C/C++, non sempre si combina perfettamente con il Python. Ma dal momento che ciò che fa realmente DDD è inviare i vostri comandi cliccati a PDB, come potete constatare nella Console di DDD, qualsiasi cosa che DDD non possa fare per voi, potete battere i comandi PDB direttamente nella Console.

## A.4 Scovare gli errori con Winpdb

Il debugger Winpdb (**www.digitalpeers.com/pythondebugger/**)[37] è molto buono. Fra le altre cose, può essere usato per scovare gli errori in codice con thread, codice basato su curses e così via, cosa che molti altri debugger non fanno. Winpdb è un programma frontale GUI al testuale RPDB2, che si trova nel medesimo pacchetto. Ho un manuale di entrambi su **http://heather.cs.ucdavis.edu/~matloff/winpdb.html**.

## A.5 Scovare gli errori con Eclipse

Personalmente non gradisco gli ambienti di sviluppo integrati (Integrated Development Environment, o IDE). Essi tendono ad essere piuttosto lenti da caricare, spesso non mi consentono di utilizzare il mio editor di testi preferito[38] e dal mio punto di vista non aggiungono molta funzionalità. Tuttavia, se siete dei fan degli IDE, vi suggerisco Eclipse, di cui ho un manuale su **http://heather.cs.ucdavis.edu/~matloff/eclipse.html**. Il mio manuale è più completo della maggioranza, permettendovi di evitare i "grattacapi" e di avere una navigazione scorrevole.

## A.6 Alcuni aiuti interni a Python per la ricerca degli errori

Esistono svariate funzioni integrate nel Python che potreste trovare utili durante il processo di ricerca degli errori.

### A.6.1 L'attributo `dict`

Richiamate quelle istanze delle classi che sono implementate come dizionari. Se avete una istanza di classe **i**, potete vedere il dizionario che ne è la sua implementazione tramite **i._dict_**. Ciò vi mostrerà i valori di tutte le variabili appartenenti alla classe.

### A.6.2 La funzione `id()`

Qualche volta è utile conoscere l'effettivo indirizzo di memoria di un oggetto. Per esempio, potreste avere due variabili che ritenete puntare allo stesso oggetto, ma non ne siete sicuri. Il metodo **id()** vi fornirà l'indirizzo dell'oggetto. Per esempio:

```
>>> x = [1,2,3]
>>> id(x)
-1084935956
>>> id(x[1])
```

37 No, non è solo per macchine Microsoft Windows, a dispetto del nome.

38 Uso vim, ma il punto importante è che voglio usare lo stesso editor per tutte le mie attività – programmazione, scrittura, posta elettronica, sviluppo di siti web, ecc...

```
137809316
```

(Non preoccupatevi dell'indirizzo “negativo”, che riflette solo il fatto di essere così alto da risultare "negativo", visto come numero intero complemento a 2).

# B - Documentazione online

## B.1 La funzione dir()

Esiste una funzione molto utile **dir()** che può essere impiegata per ottenere una rapida visione di ciò di cui è composto un dato oggetto o funzione. Dovreste usarla spesso.

Per illustrarla, nell'esempio in Sezione 10.1 supponiamo di fermarci alla linea

```
print "il numero di file di testo aperti è", textfile.ntfiles
```

Poi potremmo verificare un paio di cose con **dir()**, diciamo:

```
(Pdb) dir()
['a', 'b']
(Pdb) dir(textfile)
['__doc__', '__init__', '__module__', 'grep', 'wordcount', 'ntfiles']
```

Quando avviate la prima volta Python, si caricano varie cose. Vediamo:

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FloatingPointError',
'FutureWarning', 'IOError', 'ImportError', 'IndentationError', 'IndexError', 'KeyError',
'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError', 'None',
'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'OverflowWarning',
'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError', 'RuntimeWarning', 'StandardError',
'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',
'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',
'UnicodeTranslateError', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '_',
'__debug__', '__doc__', '__import__', '__name__', 'abs', 'apply', 'basestring', 'bool', 'buffer',
'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright', 'credits',
'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter',
'float', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int',
'intern', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'long', 'map', 'max',
'min', 'object', 'oct', 'open', 'ord', 'pow', 'property', 'quit', 'range', 'raw_input', 'reduce', 'reload',
'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum',
'super', 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

Orbene, c'è un elenco di tutte le funzioni integrate ed altri attributi per voi!

Desiderate sapere quali funzioni ed altri attributi sono associati ai dizionari?

```
>>> dir(dict)
['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__', '__doc__', '__eq__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__setitem__', '__str__', 'clear', 'copy', 'fromkeys', 'get', 'has_key', 'items', 'iteritems',
'iterkeys', 'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

Supponiamo di voler trovare quali metodi ed attributi sono associati a stringhe. Come menzionato nella Sezione 7.2.3, le stringhe ora sono una classe integrata in Python, cosicché possiamo semplicemente battere

```
>>> dir(string)
```

Ma possiamo utilizzare qualsiasi oggetto della stringa:

```
>>> dir('')
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__',
'__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__
rmod__', '__rmul__', '__setattr__', '__str__', 'capitalize', 'center', 'count', 'decode', 'encode',
```

```
'endswith', 'expandtabs', 'find', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'replace', 'rfind', 'rindex', 'rjust',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
```

## B.2 La funzione help()

Per esempio, scopriamo tutto sul metodo **pop()** per le liste:

```
>>> help(list.pop)

Help on method_descriptor:

pop(...)
        L.pop([index]) -> item -- remove and return item at index (default
last)
(END)
```

E il metodo **center()** per le stringhe:

```
>>> help(''.center)
Help on function center:

center(s, width)
        center(s, width) -> string

        Return a center version of s, in a field of the specified
        width. padded with spaces as needed.
        The string is never truncated.
```

Battete 'q' per uscire dall'impaginatore di help.

Potete ottenere informazioni anche usando **pydoc** sulla linea di comando di Unix, ad es.

```
% pydoc string.center
[...same as above]
```

## B.3 PyDoc

I metodi soprastanti per ottenere aiuti erano per l'utilizzo del modo interattivo di Python. Al di fuori di tale modo, in una shell di OS, potete ottenere le medesime informazioni da PyDoc. Per esempio,

```
pydoc sys
```

vi darà informazioni sul modulo **sys**.

Per moduli al di fuori della normale distribuzione di Python, accertatevi che si trovino nel vostro percorso di ricerca di Python ed assicuratevi di mostrare la sequenza "punto", cioè

```
pydoc u.v
```

# C - Mettere tutte le variabili globali in una classe

Come menzionato nella Sezione 5, invece di usare la parola chiave **global**, potremmo trovare più chiaro o maggiormente organizzato raggruppare tutte le nostre variabili globali in una classe. Qui, nel file **tmeg.py**, c'è come dovremmo fare ciò per modificare l'esempio di quella sezione, **tme.py:**

```
1 # legge nel file di testo quel nome che è specificato nella linea di comando
2  # e restituisce il nuero di linee e parole
3
4 import sys
5
6 def checkline():
7       glb.linecount += 1
8       w = glb.l.split()
9       glb.wordcount += len(w)
10
11 class glb:
12      linecount = 0
13      wordcount = 0
14      l = []
15
16 f = open(sys.argv[1])
17 for glb.l in f.readlines():
18      checkline()
19 print glb.linecount, glb.wordcount
```

Osservate che, quando il il programma viene caricato la prima volta, la classe **glb** verrà eseguita, anche prima dell'avvio di **main()**.

# D – Sguardo alla macchina virtuale di Python

Uno può ispezionare il codice della macchina virtuale di Python per un programma. Per il programma **srvr.py** in http://heather.cs.ucdavis.edu/~matloff/Python/PyThreads.pdf, ho fatto quanto segue:

Avviando Python in modo interattivo, ho importato per prima cosa il modulo **dis** ("disassembler"). Poi ho importato il programma battendo

```
import srvr
```

(Per primo ho dovuto aggiungere il solito codice **if __name__ == '__main__'**, in modo che il programma non andasse in esecuzione prima dell'importazione.

Poi ho lanciato

```
>>> dis.dis(srvr)
```

Come leggete il codice? Potete ottenere un elenco delle istruzioni della macchina virtuale Python in *Python: the Complete Reference*, di Martin C. Brown, pub. by Osborne, 2001. Ma se avete delle esperienze di linguaggio assembly, potreste probabilmente indovinare in ogni modo cosa fa il codice.